Máster Universitario en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO DE FIN DE MASTER

Construcción automática de grafos RDF a partir de frases en lenguaje natural para la integración de bases de datos heterogéneas

Autor: Pablo Toloza

Directores: Miguel García Remesal

MADRID, JULIO DE 2014

Dedicatoria

A mi familia y amigos que han apoyado durante mis estudios.

Resumen (Castellano)

El presente trabajo desarrolla un servicio REST que transforma frases en lenguaje natural a grafos RDF. Los grafos generados son grafos dirigidos, donde los nodos se forman con los sustantivos o adjetivos de las frases, y los arcos se forman con los verbos.

Se utiliza dentro del proyecto p-medicine para dar soporte a las siguientes funcionalidades:

Búsquedas en lenguaje natural: actualmente la plataforma p-medicine proporciona un interfaz programático para realizar consultas en SPARQL. El servicio desarrollado permitiría generar esas consultas automáticamente a partir de frases en lenguaje natural

Anotaciones de bases de datos mediante lenguaje natural: la plataforma p-medicine incorpora una herramienta, desarrollada por el Grupo de Ingeniería Biomédica de la Universidad Politécnica de Madrid, para la anotación de bases de datos RDF. Estas anotaciones son necesarias para la posterior traducción de las bases de datos a un esquema central. El proceso de anotación requiere que el usuario construya de forma manual las vistas RDF que desea anotar, lo que requiere mostrar gráficamente el esquema RDF y que el usuario construya vistas RDF seleccionando las clases y relaciones necesarias. Este proceso es a menudo complejo y demasiado difícil para un usuario sin perfil técnico. El sistema se incorporará para permitir que la construcción de estas vistas se realice con lenguaje natural.

Summary (English)

The present work develops a REST service that transforms natural language sentences to RDF degrees. Generated graphs are directed graphs where nodes are formed with nouns or adjectives of phrases, and the arcs are formed with verbs.

Used within the p-medicine project to support the following functionality:

Natural language queries: currently the p-medicine platform provides a programmatic interface to query SPARQL. The developed service would automatically generate those queries from natural language sentences.

Memos databases using natural language: the p-medicine platform incorporates a tool, developed by the Group of Biomedical Engineering at the Polytechnic University of Madrid, for the annotation of RDF data bases. Such annotations are necessary for the subsequent translation of databases to a central scheme. The annotation process requires the user to manually construct the RDF views that he wants annotate, requiring graphically display the RDF schema and the user to build RDF views by selecting classes and relationships. This process is often complex and too difficult for a user with no technical background. The system is incorporated to allow the construction of these views to be performed with natural language.

Tabla de Contenidos

Dedicatoria ii
Resumen (Castellano) iv
Summary (English) vi
Tabla de Contenidos viii
Listado de Figuras x
Listado de Tablas xii
Introducción y Objetivos 1
Objetivos 4
Estado del Arte 7
Servicios Rest 7
Propiedades Arquitectónicas 7
Restricciones Arquitectónicas 8
Concepto 10
Aplicación a Servicios Web 10
Python 11
Calidad de Software 12
Productividad de Programador 12
¿Python es un lenguaje de scripting? 12
¿Qué se puede hacer con Python? 13
¿Cuáles son las fortalezas técnicas de Python? 13
Flask 14
Configuración y Convenciones 14
Procesamiento de lenguaje Natural 15
Etiquetado Gramatical 15
Brill Tagger 15
Brown Corpus 15
Etiquetas de partes de la oración que utiliza Brown 16
NLTK 20
NLTK Taggers 21
Metamap 26
UMLS 26
Propósito del UMLS 26

Red Semantica 27
Evaluación de Riesgos 31
Desarrollo 33
Planificación 33
Creación de entorno de trabajo 34
Instrucciones de Instalación 34
Prácticas con Python, Flask y NLTK 37
Vista de Componentes 40
Creación del servicio Web 41
Procesamiento de Información 43
Resultados 47
Conclusiones 49
Líneas futuras 51
Bibiliografía 53
Anexos 55
Código Fuente Proyecto 55
Service/appService.py 55
Service/my_taggers.py 56
Service/templates/boot.html 60
Service/templates/paths.xml 63
SRInteractive.java 64

Listado de Figuras

Figura 1 Vista general....2
Figura 2 Consorcio p-medicine....4
Figura 3 Red semántica....28
Figura 4 Jerarquía de relaciones....29
Figura 5 Red semántica y relaciones....30
Figura 6 Planificación en Trello.com....33
Figura 7 Ejemplo de detalle de tareas....34
Figura 8 Código de práctica Flask y Python....37
Figura 9 metodo GET del servicio....37
Figura 10 ver solo una tarea....38
Figura 11 método POST del servicio....38
Figura 12 Prácticas con NLTK....38
Figura 13 Prácticas con NLTK....39
Figura 14 Precisión de los etiquetadores....39
Figura 15 Componentes....40
Figura 16 Servicio de interfaz de pruebas....41
Figura 17 Interfaz de pruebas....41
Figura 18 Urls aceptadas....41
Figura 19 recursos no encontrados....42
Figura 20 Entrenamiento del etiquetador....43
Figura 21 Almacenamiento del etiquetador....44
Figura 22 Preparación de parametros Metamap....44
Figura 23 Llamada al servicio Metamap....45
Figura 24 Template para la generación de XML....46
Figura 25 Ejemplo de llamada al servicio - Metodo GET....47
Figura 26 Fallo de los etiquetadores....48
Figura 27 Tiempo de respuesta de Metamap....49

Listado de Tablas

Tabla 1 Módulos de NLTK.....21
Tabla 2 Objetivos de diseño de NLTK.....21

Introducción y Objetivos

El cuidado de la salud ha sido tradicionalmente una disciplina reactiva. Sin embargo, en los últimos años, la medicina ha experimentado un cambio revolucionario, transformando gradualmente este tratamiento reactivo en un enfoque preventivo y predictivo. Este cambio ha sido posible debido a recientes logros en disciplinas relacionadas con la medicina, tales como la biología molecular, la bioinformática, la biología de sistemas y el aumento de la capacidad de los ordenadores para manejar grandes cantidades de datos heterogéneos, lo que lleva a una mejor comprensión de las enfermedades.

Estos avances están transformando la medicina tradicional en una medicina personalizada, donde el diagnóstico y tratamiento de las enfermedades se basan en el uso integrado de datos clínicos y genómicos exclusivos de cada paciente. Este nuevo paradigma permite el diseño de medicamentos y tratamientos personalizados para cada paciente concreto, potenciando los beneficios de dichas terapias y a la vez minimizando los efectos secundarios.

Es en este contexto, surge el proyecto p-Medicine, un proyecto integrado de 4 años de duración, financiado por la Comisión Europea, cuyo objetivo es desarrollar una infraestructura tecnológica (IT) para dar soporte a la medicina personalizada. En el proyecto p-medicine, 19 socios de 9 países europeos y Japón se han dedicado a crear, apoyar y sostener las nuevas tecnologías del conocimiento e innovación para superar los problemas actuales en la investigación clínica y allanar el camino para lograr terapias más personalizadas.

En líneas generales, el proyecto p-medicine está estructurado en dos tipos de escenarios:

1. Integración de grandes conjuntos de datos, pseudo-anónimos y provenientes de múltiples fuentes de datos, que se utilizan para el desarrollo de herramientas de VPH;
2. Uso de los datos asociados a un paciente concreto para ejecutar un flujo de trabajo de simulación en apoyo a un proceso individual de toma de decisiones clínicas.

La infraestructura de p-medicine consiste en una IT y una infraestructura de investigación clínica que se interconectarán para dar soporte a los escenarios descritos con anterioridad. La arquitectura del sistema será modular, de manera tal que la transición hacia la infraestructura desarrollada en el proyecto p-medicine podrá realizarse de manera gradual.

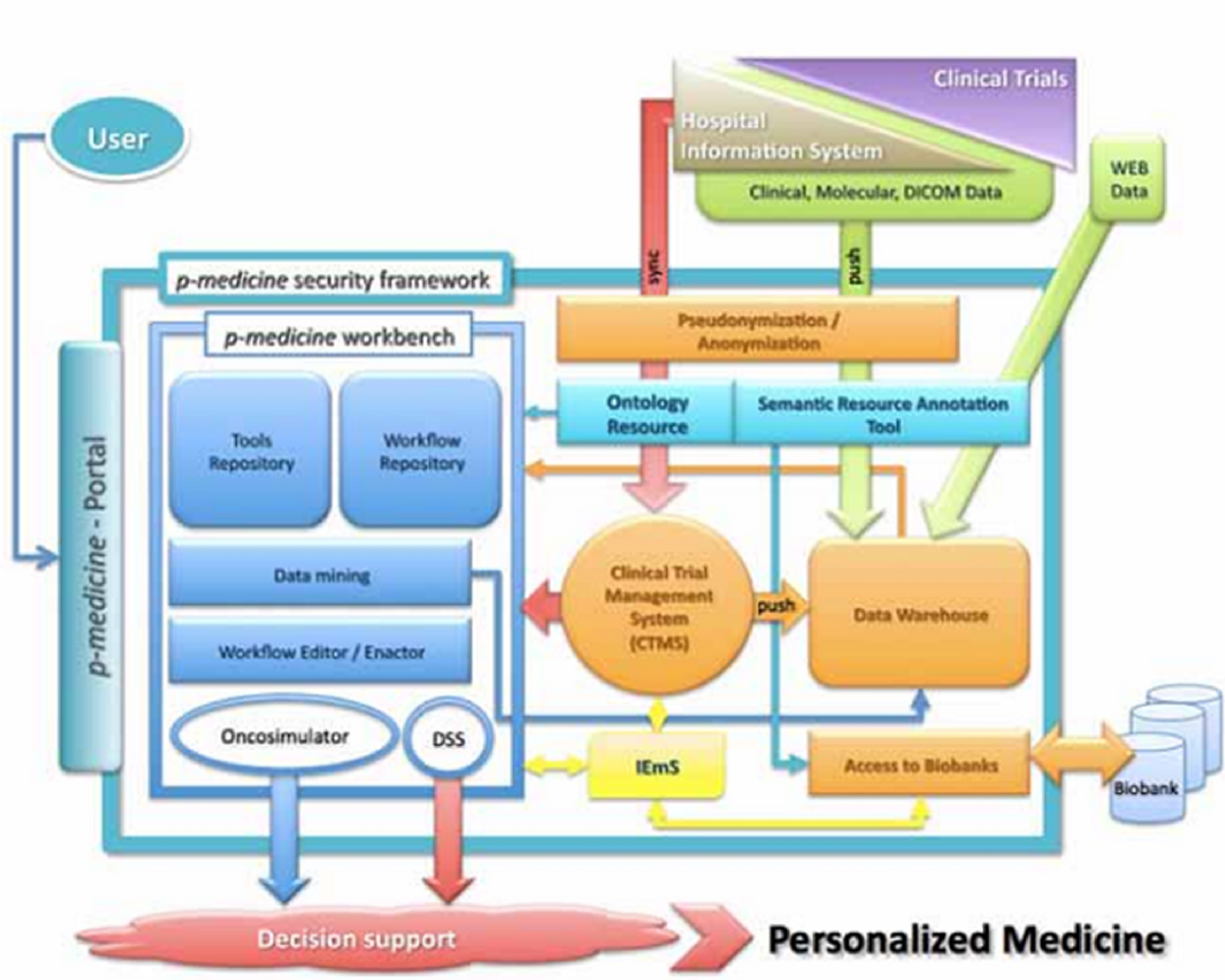

Figura 1 Vista general

Respecto al tema de la protección de datos, se garantizará que las políticas de privacidad, no discriminación y acceso estarán alineadas para maximizar la protección de los pacientes. El marco legal y ético que se aplicará en p-medicine abordará específicamente estas condiciones. Asimismo, la infraestructura de IT, en sí garantizará la protección y la seguridad de los datos, dado que no se almacenarán datos personales de los pacientes.

Los datos pseudo-anónimos y las herramientas desarrolladas serán alojados en repositorios, los cuales será accesibles por investigadores y usuarios finales en virtud de contratos vinculantes. El repositorio de datos se utilizará para la realización de simulaciones de VPH y para llevar a cabo las pruebas de las herramientas desarrolladas, para las cuales la normalización y la interoperabilidad semántica es una cuestión clave.

Es asimismo un objetivo de gran importancia, que las herramientas desarrolladas, cumplan con los requisitos para ser utilizadas en ensayos clínicos internacionales de gran escala. Concretamente, los ensayos clínicos incorporados en p-medicine se enfocan en tres tipos de cáncer y tres aspectos diferentes del marco de trabajo desarrollado:

- El ensayo del tumor de Wilms se utilizará para emplear las herramientas recién desarrolladas y validadas de p-medicine. El ensayo también proporciona datos

para el Oncosimulator (simulador de crecimiento de tumores) que pondrá a prueba un escenario específico del tumor de Wilms;

- Se utilizarán los ensayos de cáncer de mama para la validación de las herramientas de toma de decisiones y la adquisición de datos, intercambio y análisis. Además, el ensayo de cáncer de mama neoadyuvante de fase II farmacodinámico se extenderá a las herramientas de VPH;
- Los ensayos leucemia y de cáncer de mama en fase farmacodinámica neoadyuvante II se utilizarán para ejecutar biología de sistemas y escenarios dinámicos posgenómica para encontrar los factores de riesgo individuales para la toma de decisiones y para validar los modelos propuestos.

Los datos de los sistemas de información clínicos estarán disponibles por un modelo "push" donde los propietarios de los datos iniciarán las transferencias. El acceso a los biobancos ayudará a responder a las preguntas de investigación sin llevar a cabo nuevos ensayos. Permitir a los pacientes decidir en cualquier momento qué tipo de investigación se puede hacer con sus datos y su biomaterial refuerza la potestad del paciente sobre los mismos.

En el proyecto p-medicine se pondrá especial atención, en todas sus actividades de investigación, para incentivar la creación de colaboraciones e interacciones con otros proyectos e iniciativas, y para poder aprender del trabajo ya realizado. En este sentido, el proyecto FP6 ACGT (Advancing ClinicoGenomic Trials in cancer) se presenta como referencia y base de p-medicine. La interacción ya ha comenzado con éxito con otros proyectos en curso del FP7 como TUMOR (Transatlantic tumor Model Repositories), CONTRACT (CONsent in a TRial And Care EnvironmenT) y, principalmente, con VPH-Share (Virtual Physiological Human: Sharing for Healthcare).

Una estrecha colaboración se establecerá con las partes interesadas en la asistencia sanitaria, incluidos los investigadores de TI y la industria, para darles la oportunidad de aprender sobre el entorno, las herramientas y servicios de p-medicine, a través de herramientas de e-learning y talleres de capacitación.

Las respuestas de los investigadores y los especialistas de la industria serán de interés vital para mejorar y optimizar estas herramientas y servicios en los próximos años de duración del proyecto.

Figura 2 Consorcio p-medicine

Objetivos

En el contexto del proyecto p-medicine descripto anteriormente, el presente trabajo tiene por objetivo desarrollar un servicio para dar soporte a las siguientes funcionalidades:

Búsquedas en lenguaje natural: actualmente la plataforma p-medicine proporciona un interfaz programático para realizar consultas en SPARQL. El servicio desarrollado permitiría generar esas consultas automáticamente a partir de frases en lenguaje natural del usuario.

Anotaciones de bases de datos mediante lenguaje natural: la plataforma p-medicine incorpora una herramienta, desarrollada por el Grupo de Informática Biomédica de la UPM, para la anotación de bases de datos RDF. Estas anotaciones son necesarias para la posterior traducción de las bases de datos a un esquema central. El proceso de anotación requiere que el usuario construya de forma manual las vistas RDF que desea anotar, lo que requiere mostrar gráficamente el esquema RDF y que el usuario construya vistas RDF seleccionando las clases y relaciones necesarias. Este proceso es a menudo complejo y demasiado difícil para un usuario sin perfil técnico. El servicio se incorporará para permitir que la construcción de estas vistas se realice con lenguaje natural.

El servicio que se desarrollará transformará frases en lenguaje natural en grafos RDF. Los grafos a generar son grafos dirigidos, donde los nodos se forman con los sustantivos o adjetivos de las frases, y los arcos se forman con los verbos. Para esto, el servicio desarrollado, interpretará las frases en lenguaje natural y generará un grafo utilizando

notación XML que luego será utilizado por otras herramientas que están fuera del alcance de este trabajo. El servicio desarrollado deberá ser escalable y modular, ya que ésta es la filosofía del proyecto p-medicine.

Estado del Arte

Servicios Rest

REpresentational State Transfer (REST) es un estilo de arquitectura de software que consta de un conjunto coordinado de restricciones arquitectónicas aplicadas a los componentes, conectores y elementos de datos, dentro de un sistema distribuido. REST pasa por alto los detalles de la implementación de los componentes y la sintaxis de protocolo con el fin de centrarse en las funciones de los componentes, las restricciones sobre su interacción con otros componentes, y su interpretación de los elementos de datos significativos.

El término "transferencia de estado representacional" fue introducido y definido en 2000 por Roy Fielding en su tesis doctoral en la Universidad de California en Irvine. REST se ha aplicado para describir la arquitectura web deseada, para identificar los problemas existentes, para comparar soluciones alternativas, y para asegurarse que las extensiones de protocolo no violarían las restricciones básicas que hacen que la web sea exitosa. Fielding utilizó REST para diseñar HTTP 1.1 y los identificadores de recursos uniformes (URI). El estilo arquitectónico REST también se aplica al desarrollo de servicios web como alternativa a otras especificaciones de computación distribuida como SOAP.

Propiedades Arquitectónicas

Las propiedades arquitectónicas inducidas por las restricciones del estilo arquitectónico REST son:

- Rendimiento: Las interacciones entre componentes puede ser el factor dominante en el rendimiento percibido por el usuario y la eficiencia de la red.
- Escalabilidad para soportar un gran número de componentes e interacciones entre los mismos.
- La simplicidad de las interfaces.
- Facilidad de cambio de componentes para satisfacer las necesidades cambiantes (incluso mientras se ejecuta la aplicación).
- Visibilidad de la comunicación entre los componentes de los agentes de servicio.
- Portabilidad de los componentes moviendo código de programa con los datos.
- La fiabilidad es la resistencia a fallos a nivel del sistema en presencia de fallos en los componentes, conectores, o datos.

Fielding describe el efecto de REST en la escalabilidad de este modo:

El interés de REST en la separación cliente-servidor, simplifica la implementación de componentes, reduce la complejidad de la semántica de conectores, mejora la eficacia de la optimización del rendimiento y aumenta la escalabilidad de los componentes de servidor puros. Las limitaciones del sistema en capas permiten que se introduzcan intermediarios -proxies, gateways y servidores de seguridad- en varios puntos de la comunicación sin necesidad de cambiar las interfaces entre los componentes, lo que les permite ayudar en la traducción de comunicaciones o mejorar el rendimiento a través del almacenamiento en caché de gran escala, y compartida. REST permite el procesamiento intermedio al restringir los mensajes para sólo sean auto descriptivos: la interacción es sin estado entre las solicitudes, los métodos estándar y los tipos de medios se utilizan para indicar la semántica y el intercambio de información y las respuestas indican explícitamente si se almacenan en caché.

Restricciones Arquitectónicas

Las propiedades arquitectónicas de REST se consiguen mediante la aplicación de las limitaciones de interacción específicas a los componentes, conectores y elementos de datos. Las limitaciones REST son:

Cliente Servidor:

Una interfaz uniforme separa clientes de servidores. Esta separación de las responsabilidades significa que, por ejemplo, los clientes no son responsables del almacenamiento de datos, que sigue siendo interno para cada servidor, de manera que se mejora la portabilidad de código de cliente. Los servidores no tienen que ver con la interfaz de usuario o el estado del usuario, por lo que los servidores pueden ser más simples y más escalables. Los servidores y los clientes también pueden ser sustituidos y ser desarrollados de forma independiente, siempre que la interfaz entre ellos no se altere.

Stateless (sin Estado)

La comunicación entre cliente y servidor se ve limitada además porque ningún contexto del cliente se almacena en el servidor entre las peticiones. Cada solicitud de cualquier cliente contiene toda la información necesaria para atender la solicitud, y el estado de sesión se lleva a cabo en el cliente. Es importante tener en cuenta que el estado de sesión puede ser transferido por el servidor a otro servicio, como una base de datos para mantener un estado persistente durante un período y permitir la autenticación. El cliente comienza a enviar solicitudes cuando está listo para hacer la transición a un nuevo estado. Mientras que una o más solicitudes son excepcionales, el cliente se considera que está en transición. La representación de cada estado de la aplicación contiene enlaces que pueden utilizarse la próxima vez que el cliente decide iniciar una nueva transición de estado.

Cacheable

Como en la Web, los clientes pueden almacenar en caché las respuestas. Las respuestas deben, por lo tanto, de manera implícita o explícita, definirse como almacenables en caché, o no, para evitar que los clientes reutilicen datos obsoletos o inadecuados en

respuesta a las nuevas solicitudes. El Caching bien gestionado elimina parcial o completamente algunas interacciones cliente-servidor, mejorando aún más la escalabilidad y el rendimiento.

Sistemas en Capas

Un cliente no puede decir normalmente si está conectado directamente al servidor final, o a un intermediario a lo largo del camino. Los servidores intermediarios pueden mejorar la escalabilidad del sistema, permitiendo el balanceo de carga y proporcionando caches compartidas. También pueden hacer cumplir las políticas de seguridad.

Código bajo demanda

Los servidores pueden ampliar o personalizar la funcionalidad de un cliente mediante la transferencia de código ejecutable de forma temporal. Ejemplos de esto pueden incluir componentes compilados tales como applets de Java y scripts del lado del cliente como JavaScript. "Código bajo demanda" es la única restricción opcional de la arquitectura REST.

Interface Uniforme

La restricción de interfaz uniforme es fundamental para el diseño de cualquier servicio REST. La interfaz uniforme y simplifica, desacopla la arquitectura, que permite a cada parte evolucionar de forma independiente. Los cuatro principios que guían esta interfaz son:

Identificación de recursos

Los recursos individuales se identifican en las solicitudes, por ejemplo utilizando las URI en los sistemas REST basados en Web. Los recursos propios son conceptualmente separados de las representaciones que se devuelven al cliente. Por ejemplo, el servidor no envía su base de datos, sino que tal vez, algo de HTML, XML o JSON.

Manipulación de recursos a través de representaciones

Cuando un cliente tiene una representación de un recurso, incluyendo cualquier metadato adjunto, tiene suficiente información para modificar o eliminar el recurso.

Mensajes autodescriptivos

Cada mensaje incluye suficiente información para describir la forma de procesar el mensaje. Por ejemplo, qué analizador invocar podría ser especificado por medio de un tipo de medio de Internet (antes conocido como un tipo MIME). Las respuestas también indican explícitamente su almacenamiento en caché.

Hipermedia como el motor del estado de aplicación

Los clientes hacen las transiciones de estado sólo a través de acciones que se identifican de forma dinámica dentro de hipermedia por el servidor (por ejemplo, por medio de hipervínculos dentro de hipertexto). A excepción de simples puntos fijos de entrada a la demanda, un cliente no asume que cualquier acción en particular está disponible para todos los recursos particulares más allá de los descritos en las representaciones previamente recibidos del servidor.

Las aplicaciones que se ajusten a las restricciones REST que se describen en esta sección pueden considerarse como "RESTful". Si un servicio viola cualquiera de las restricciones necesarias, no se puede considerar RESTful.

El cumplimiento de estas restricciones conforman el estilo arquitectónico REST, y permite a cualquier tipo de sistema hipermedia distribuido tener propiedades emergentes deseables, tales como el rendimiento, la escalabilidad, la sencillez, la facilidad de cambio, visibilidad, portabilidad y fiabilidad.

Concepto

La Transferencia de estado representacional está destinada a evocar una imagen de cómo una aplicación web bien diseñada se comporta: se presenta con una red de páginas web (un estado de la máquina virtual), el usuario avanza a través de una aplicación mediante la selección de enlaces (transiciones de estado), lo que resulta en la página siguiente (que representa el siguiente estado de la aplicación) que se transfiere al usuario y se muestra para su uso.

REST fue descrita inicialmente en el contexto de HTTP, pero no se limita a ese protocolo. Las arquitecturas REST pueden basarse en otros protocolos de la capa de aplicación, si es que estos ya proporcionan un vocabulario rico y uniforme para las aplicaciones basadas en la transferencia de la representación significativa estado. Las aplicaciones RESTful maximizan el uso de la interfaz existente, bien definida y otras capacidades incorporadas proporcionadas por el protocolo de red elegido, y minimizan la adición de nuevas características específicas de la aplicación en la parte superior de la misma.

Aplicación a Servicios Web

Los API de servicios Web que se adhieren a las restricciones de REST se llaman RESTful. Los APIs REST definen estos aspectos:

- URI base, como http://example.com/resources/
- un tipo de medio de Internet para los datos. Esto es a menudo JSON, pero puede ser cualquier otro tipo de medios de Internet válido (por ejemplo, XML, Atom, imágenes, etc)
- métodos HTTP estándar (por ejemplo, GET, PUT, POST o DELETE)
- enlaces de hipertexto para referencia del Estado
- enlaces a recursos de referencia relacionados

Los métodos PUT y DELETE son métodos idempotente. El método GET es un método seguro (o nullipotent), lo que significa que llamarlo no produce efectos secundarios.

A diferencia de los servicios web basados en SOAP, no existe un estándar "oficial" para las API RESTful. Esto se debe a que REST es un estilo arquitectónico, a diferencia de SOAP, que es un protocolo. A pesar de que REST no es un estándar, una implementación REST puede utilizar estándares como HTTP, URI, XML, etc

Python

Python es un lenguaje de programación fácil de aprender y potente. Cuenta con estructuras de datos de alto nivel eficientes y un enfoque simple pero eficaz para programación orientada a objetos. La elegante sintaxis de Python y tipado dinámico, junto con su naturaleza interpretada, lo convierten en un lenguaje ideal para secuencias de comandos y el desarrollo rápido de aplicaciones en muchas áreas y en la mayoría de las plataformas.

El intérprete de Python y la extensa biblioteca estándar están disponibles gratuitamente en formato fuente o binario para todas las plataformas desde el sitio Web de Python, y pueden ser distribuidos libremente. El mismo sitio contiene también distribuciones y enlaces a muchos módulos libres de Python de terceros, programas y herramientas y documentación adicional.

El intérprete de Python se extiende fácilmente con nuevas funciones y tipos de datos implementados en C o C++ (u otros idiomas que se pueden llamar desde C). Python también es adecuado como un lenguaje de extensión para aplicaciones personalizables.

Python es simple de usar, pero es un lenguaje de programación real, ofreciendo mucho más estructura y apoyo para los programas grandes que lo que pueden ofrecer los scripts o archivos por lotes. Por otro lado, Python también ofrece mucho más para la comprobación de errores que C, y al ser un lenguaje de muy alto nivel, cuenta con tipos de datos de alto nivel incluidos, como matrices y diversos diccionarios. Debido a sus tipos de datos más generales, Python es aplicable a un dominio del problema mucho más grande que Awk o incluso Perl, muchas cosas son al menos tan fácil en Python como en esos lenguajes.

Python permite dividir el programa en módulos que se pueden reutilizar en otros programas Python. Viene con una gran colección de módulos estándar que se pueden utilizar como base, o como ejemplos para empezar a aprender a programar en Python. Algunos de estos módulos ofrecen cosas como I/O, llamadas al sistema, sockets y hasta interfaces a kits de herramientas de interfaz gráfica de usuario como Tk.

Python es un lenguaje interpretado, lo que le puede ahorrar un tiempo considerable durante el desarrollo del programa, dado que no es necesario compilar ni enlazar. El intérprete se puede utilizar de forma interactiva, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones durante el desarrollo de un programa.

Python permite que los programas sean escritos de manera compacta y legible. Los programas escritos en Python son típicamente mucho más cortos que los programas en C, C++, Java o equivalentes, por varias razones:

- los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción;
- agrupamiento de sentencias se realiza mediante la indentación en lugar de comienzo y finalización entre paréntesis;

- las declaraciones de variables o argumentos no son necesarios.

Python es extensible: si se sabe cómo programar en C es fácil de agregar una nueva función o módulo integrado para el intérprete, ya sea para realizar operaciones críticas a máxima velocidad, o para vincular los programas de Python a las bibliotecas que sólo pueden estar disponibles en forma binaria. Es relativamente sencillo enlazar el intérprete de Python en una aplicación escrita en C y utilizarlo como una extensión o un lenguaje de comandos para esa aplicación.

Calidad de Software

Por diseño, Python implementa una sintaxis deliberadamente simple y de fácil lectura y un modelo de programación muy coherente. Esto hace que el lenguaje sea más fácil de aprender, entender y recordar. En la práctica, los programadores de Python no tienen que referirse constantemente a los manuales durante la lectura o la escritura de código.

Por filosofía, Python adopta un enfoque un tanto minimalista. Esto significa que aunque generalmente haya múltiples maneras de lograr una tarea de codificación, por lo general hay sólo una manera obvia, algunas alternativas menos obvias, y un pequeño conjunto de interacciones coherentes en todas partes en el lenguaje. Por otra parte, Python no toma decisiones arbitrarias por los usuarios; cuando las interacciones son ambiguas, se prefiere la intervención explícita sobre "la magia." En la forma de pensar de Python, explícito es mejor que implícito, y simple es mejor que complejo.

Más allá de estos temas de diseño, Python incluye herramientas tales como módulos y programación orientada a objetos que, promueven la reutilización del código.

Productividad de Programador

Durante el gran boom de Internet de la década del ‘90, era difícil encontrar suficientes programadores para implementar proyectos de software; Se pedía a los desarrolladores implementar sistemas tan rápido como Internet evolucionaba. En épocas de crisis y recesión económica, el panorama cambió. A los equipos de desarrollo se les pedía realizar las mismas tareas con aún menos personas.

En ambos escenarios, Python ha brillado como una herramienta que permite a los programadores hacer más cosas con menos esfuerzo. Está optimizado para la velocidad de desarrollo, ya que su sintaxis es simple, el tipado es dinámico, y no es necesario compilar cada vez que se hay una modificación. Como resultado se obtiene que Python normalmente aumenta productividad de los desarrolladores mucho más allá de los niveles soportados por los lenguajes tradicionales. Esas son buenas noticias tanto en los buenos como en los malos tiempos.

¿Python es un lenguaje de scripting?

Python es un lenguaje de programación de propósito general que se aplica a menudo en scripting. Se define comúnmente como un lenguaje de scripting orientado a objetos Si se presiona por una sola línea, se diría que Python es probablemente mejor conocido como un lenguaje de programación de propósito general que combina los paradigmas procedural, funcional y orientado a objetos.

Sin embargo, el término "scripting" parece haberse pegado a Python, tal vez como un contraste con el mayor esfuerzo de programación requerido por algunas otras herramientas. Por ejemplo, la gente suele usar la palabra "script" en lugar de "programa" para describir un archivo de código Python.

¿Qué se puede hacer con Python?

Además de ser un lenguaje de programación bien diseñado, Python es útil para llevar a cabo las tareas del mundo real, la clase de cosas que los desarrolladores hacen día tras día. Es comúnmente usado en una variedad de dominios, como herramienta para las secuencias de comandos de otros componentes y la implementación de programas independientes. De hecho, como lenguaje de propósito general, las funciones de Python son prácticamente ilimitadas: se puede utilizar para todo, desde desarrollo de sitios web hasta juegos y robótica.

¿Cuáles son las fortalezas técnicas de Python?

Python es un lenguaje orientado a objetos, desde de cero. Su modelo de clases admite nociones avanzadas tales como el polimorfismo, la sobrecarga de operadores, y la herencia múltiple; sin embargo, en el contexto de la sintaxis simple y tipificación de Python, la programación orientada a objetos es muy fácil de aplicar.

Además de servir como un poderoso estructurador de código y dispositivo de reutilización, la naturaleza OOP de Python lo hace ideal como herramienta de scripting para otros lenguajes de sistemas orientados a objetos. Por ejemplo, con el código de adecuado, los programas Python pueden crear subclases (especializan) de clases implementadas en C++, Java y C #.

De igual importancia es que la programación orientada a objetos es una opción en Python; se puede ir muy lejos sin tener que convertirse en un gurú de objetos. Al igual que C++, Python admite los modos de programación tanto de procedimientos como de orientación a objetos. Sus herramientas orientadas a objetos se pueden aplicar siempre y cuando las restricciones lo permitan. Esto es especialmente útil en los modos de desarrollo tácticos, que impiden fases de diseño.

Además de sus paradigmas estructurel y de objetos, Python en los últimos años ha adquirido una función de apoyo a la programación funcional, conjunto que por la mayoría de las medidas incluye generadores, comprensiones, cierres, mapas, decoradores, lambdas de funciones anónimas y objetos de función de primera clase. Estos pueden servir como complemento y alternativa a sus herramientas de programación orientada a objetos.

Flask

Flask es un micro framework para desarrollo web en Python. "Micro" no significa que la aplicación web todo tiene que estar en un solo archivo de Python, aunque ciertamente puede. Tampoco significa que Flask carece de funcionalidad. El "micro" en “micro framework” significa que Flask tiene como objetivo mantener el núcleo simple pero extensible. Flask no toma muchas decisiones por el usuario, tales como qué base de datos utilizar pero las decisiones que si toma, como el motor de plantillas para utilizar, son fáciles de cambiar. Todo lo demás depende del usuario, de esta forma Flask puede ser todo lo que se necesita y nada de lo que no se desee.

Por defecto, Flask no incluye una capa de abstracción de base de datos, validación de formularios o cualquier otra cosa donde ya existen diferentes bibliotecas que puede manejarlo. En cambio, Flask soporta extensiones para agregar esta funcionalidad a su aplicación como si ya estuviera incluido. Numerosas extensiones proporcionan la integración de bases de datos, validación de formularios, manejo de carga, varias tecnologías de autenticación abierta, y más. Flask puede ser "micro", pero está listo para el uso en producción en una variedad de necesidades.

Configuración y Convenciones

Flask tiene muchos valores de configuración, con valores razonables por defecto, y una serie de convenciones cuando inicia. Por convención, los templates y archivos estáticos se almacenan en subdirectorios dentro árbol de código fuente de Python de la aplicación, con los nombres “templates” y “static” respectivamente. Si bien esto puede ser cambiado por lo general no tiene que hacerse.

Procesamiento de lenguaje Natural

El Procesamiento del lenguaje natural (NLP) es un campo de la informática, la inteligencia artificial, y la lingüística que se ocupa de las interacciones entre las computadoras y los lenguajes humanos (naturales). Como tal, el NLP se relaciona con el campo de la interacción persona-ordenador. Muchos desafíos en NLP implican la comprensión del lenguaje natural, es decir, permitir a los ordenadores entender el significado de entradas en lenguaje natural, y otros implican generación de lenguaje natural.

Etiquetado Gramatical

El etiquetado gramatical (POS, Part of Speech, parte de la oración) es más difícil que simplemente tener una lista de palabras y sus partes de la oración, ya que algunas palabras pueden representar más de una parte de la oración en diferentes momentos, debido a que algunas partes de la oración son complejas o no se mencionan. Esto no es raro en las lenguas naturales (a diferencia de muchos lenguajes artificiales), un gran porcentaje de las formas verbales son ambiguas. Por ejemplo, la palabra “dado”, que por lo general se piensa como simplemente un sustantivo singular, también puede ser un adjetivo.

Es muy dado a participar.

El etiquetado gramatical correcto reflejará ese "dado" que se utiliza aquí como un adjetivo, no como el sustantivo singular. El contexto gramatical es una manera de determinar esto; el análisis semántico también puede utilizarse para inferir que "es" y "muy" implican "dado" como: que tiene tendencia a algo.

En ingles normalmente se manejan 9 partes de la oración: sustantivo, verbo, artículo, adjetivo, pronombre, preposición, adverbio, conjunción e interjección. Sin embargo, hay muchas más categorías y subcategorías. Para los nombres se pueden distinguir las formas plurales, posesivas, y singulares. En muchos lenguajes de palabras también están marcadas por su "caso" (papel como sujeto, objeto, etc), el género gramatical, y así sucesivamente; mientras que los verbos se marcan por su tiempo, aspecto, y otras cosas. Los lingüistas distinguen las partes de la oración de diferentes maneras, reflejo del "sistema de etiquetado " elegido.

En las partes de la oración etiquetadas por ordenador, es típico de distinguir entre 50 y 150 partes separadas de voz para el Inglés. Por ejemplo, NN para los sustantivos comunes singulares, NNS para los sustantivos comunes plurales, NP para los nombres propios singulares.

Brill Tagger

El BrillTagger es un etiquetador especial. Por un lado, no es un etiquetador secuencial, a pesar de que hace uso de un etiquetador POS inicial. El etiquetador Brill utiliza el etiquetador POS (Part Of Speech) inicial para generar las etiquetas iniciales, y luego corrige esas etiquetas, utilizando reglas transformacionales. Estas reglas se aprenden mediante el entrenamiento del etiquetador de Brill con un entrenador y unas plantillas de reglas.

Brown Corpus

La investigación sobre el etiquetado de las partes de la oración ha estado estrechamente ligada a la lingüística de corpus. El primer corpus importante de inglés para el análisis en ordenador es el de Brown, desarrollado en la Universidad de Brown por Henry

Kucera y Nelson Francis, a mediados de la década del ‘60. Se compone de alrededor de 1.000.000 palabras de texto en Inglés, compuesto de 500 muestras de las publicaciones elegidas al azar. Cada muestra es de 2.000 o más palabras (finalizando en la primera frase, después de 2000 palabras, para que el cuerpo contenga sólo frases completas).

El Corpus de Brown fue cuidadosamente "etiquetado" con marcadores de parte de la oración durante muchos años. Una primera aproximación se hizo con un programa de Greene y Rubin, que consistía en una enorme lista hecha a mano de las categorías que podían ocurrir simultáneamente. Por ejemplo, a continuación del artículo puede haber un sustantivo, pero no puede haber un verbo. El programa consiguió un 70% de respuestas correctas. Sus resultados fueron revisados en varias ocasiones y corregidos a mano, y los usuarios posteriormente han enviaron mas correcciones, por lo que a finales de los años 70 el etiquetado era casi perfecto (habiendo algunos casos en los que incluso los humanos podrían no estar de acuerdo).

Este corpus se ha utilizado para innumerables estudios sobre frecuencias de palabra y de partes de la oración, y ha inspirado el desarrollo de otros corpus etiquetados, similares, en muchos otros idiomas. Las estadísticas obtenidas por el análisis del corpus, sirvieron de base para los sistemas de etiquetado posteriores, como CLAWS y VOLSUNGA. Sin embargo, actualmente ha sido reemplazado por corpus más grandes, como el “British National Corpus” con cerca de 100 millones de palabras.

Desde hace algún tiempo, el etiquetado de partes de la oración se considera una parte inseparable de procesamiento del lenguaje natural, porque hay ciertos casos en los que la parte correcta de la palabra no se puede decidir sin entender la semántica o incluso la pragmática del contexto. Esto es extremadamente costoso, sobre todo porque el análisis de los niveles más altos es mucho más difícil cuando deben considerarse varias posibilidades para cada palabra.

Etiquetas de partes de la oración que utiliza Brown

Etiqueta	Definición
.	sentence closer (. ; ? *)
(	left paren
)	right paren
*	not, n't
--	dash
,	comma
:	colon
ABL	pre-qualifier (quite, rather)
ABN	pre-quantifier (half, all)
ABX	pre-quantifier (both)
AP	post-determiner (many, several, next)

AT	article (a, the, no)
BE	be
BED	were
BEDZ	was
BEG	being
BEM	am
BEN	been
BER	are, art
BEZ	is
CC	coordinating conjunction (and, or)
CD	cardinal numeral (one, two, 2, etc.)
CS	subordinating conjunction (if, although)
DO	do
DOD	did
DOZ	does
DT	singular determiner/quantifier (this, that)
DTI	singular or plural determiner/quantifier (some, any)
DTS	plural determiner (these, those)
DTX	determiner/double conjunction (either)
EX	existential there
FW	foreign word (hyphenated before regular tag)
HV	have
HVD	had (past tense)
HVG	having
HVN	had (past participle)
IN	preposition
JJ	adjective
JJR	comparative adjective
JJS	semantically superlative adjective (chief, top)
JJT	morphologically superlative adjective (biggest)

MD	modal auxiliary (can, should, will)
NC	cited word (hyphenated after regular tag)
NN	singular or mass noun
NN$	possessive singular noun
NNS	plural noun
NNS$	possessive plural noun
NP	proper noun or part of name phrase
NP$	possessive proper noun
NPS	plural proper noun
NPS$	possessive plural proper noun
NR	adverbial noun (home, today, west)
OD	ordinal numeral (first, 2nd)
PN	nominal pronoun (everybody, nothing)
PN$	possessive nominal pronoun
PP$	possessive personal pronoun (my, our)
P$$	second (nominal) possessive pronoun (mine, ours)
PPL	singular reflexive/intensive personal pronoun (myself)
PPLS	plural reflexive/intensive personal pronoun (ourselves)
PPO	objective personal pronoun (me, him, it, them)
PPS	3rd. singular nominative pronoun (he, she, it, one)
PPSS	other nominative personal pronoun (I, we, they, you)
PRP	Personal pronoun
PRP$	Possessive pronoun
QL	qualifier (very, fairly)
QLP	post-qualifier (enough, indeed)
RB	adverb
RBR	comparative adverb
RBT	superlative adverb
RN	nominal adverb (here, then, indoors)
RP	adverb/particle (about, off, up)

TO	infinitive marker to
UH	interjection, exclamation
VB	verb, base form
VBD	verb, past tense
VBG	verb, present participle/gerund
VBN	verb, past participle
VBP	verb, non 3rd person, singular, present
VBZ	verb, 3rd. singular present
WDT	wh- determiner (what, which)
WP$	possessive wh- pronoun (whose)
WPO	objective wh- pronoun (whom, which, that)
WPS	nominative wh- pronoun (who, which, that)
WQL	wh- qualifier (how)
WRB	wh- adverb (how, where, when)

NLTK

NLTK es una plataforma líder para la creación de programas en Python para trabajar con datos en lenguaje natural. Proporciona interfaces fáciles de usar para más de 50 corpus y recursos léxicos como WordNet, junto con un conjunto de bibliotecas de procesamiento de textos para la clasificación, tokenización, derivación, etiquetado, análisis y razonamiento semántico.

Gracias a una guía práctica sobre introducción de fundamentos de programación junto a temas de lingüística computacional, NLTK es adecuado para los lingüistas, ingenieros, estudiantes, educadores, investigadores y usuarios de la industria por igual. NLTK está disponible para muchas plataformas. Y lo mejor de todo, es que NLTK es código abierto, un proyecto libre impulsado por la comunidad.

NLTK ha sido llamado "una maravillosa herramienta para la enseñanza y el trabajo en lingüística computacional utilizando Python" y "una biblioteca increíble jugar con el lenguaje natural."

NLTK fue creado originalmente en 2001 como parte de un curso de la lingüística computacional en el Departamento de Informática y Ciencias de la Información en la Universidad de Pennsylvania. Desde entonces se ha desarrollado y ampliado con la ayuda de docenas de contribuyentes. Ahora se ha adoptado en los cursos en decenas de universidades, y sirve como la base de muchos proyectos de investigación. En la siguiente tabla se recoge una lista de los módulos NLTK más importantes.

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	standardized interfaces to corpora and lexicons
String processing	nltk.tokenize, nltk.stem	tokenizers, sentence tokenizers, stemmers
Collocation discovery	nltk.collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill, HMM, TnT
Classification	nltk.classify, nltk.cluster	decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	regular expression, n-gram, named-entity
Parsing	nltk.parse	chart, feature-based, unification, probabilistic, dependency

Semantic interpretation	nltk.sem, nltk.inference	lambda calculus, first-order logic, model checking
Evaluation metrics	nltk.metrics	precision, recall, agreement coefficients
Probability and estimation	nltk.probability	frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	graphical concordancer, parsers, WordNet browser, chatbots
Linguistic fieldwork	nltk.toolbox	manipulate data in SIL Toolbox format

Tabla 1 Módulos de NLTK

NLTK fue diseñado con 4 objetivos primarios:

Simplicidad:	Proporcionar un marco intuitivo junto con bloques de construcción principales, dando a los usuarios un conocimiento práctico de la NLP, sin las complicaciones habituales asociadas con el procesamiento del lenguaje.
Consistencia:	Proporcionar un marco uniforme con interfaces consistentes y estructuras de datos, y nombres de métodos de fáciles de recordar.
Extensibilidad:	Proporcionar una estructura en la que los nuevos módulos de software pueden ser incluidos fácilmente, incluyendo implementaciones alternativas y enfoques que compiten con los desarrollos existentes.
Modularidad:	Proporcionar los componentes que se pueden utilizar de forma independiente sin necesidad de entender el resto del toolkit.

Tabla 2 Objetivos de diseño de NLTK

En contraste con estos objetivos hay tres no requisitos, o cualidades potencialmente útiles que se han evitado deliberadamente. En primer lugar, mientras que el conjunto de herramientas ofrece una amplia gama de funciones, no es enciclopédico; es un conjunto de herramientas, no un sistema, y seguirá evolucionando con el campo de la NLP. En segundo lugar, mientras que el conjunto de herramientas es lo suficientemente eficiente para apoyar tareas significativas, no está muy optimizado para el rendimiento en tiempo de ejecución; tales optimizaciones a menudo implican algoritmos más complejos, o implementaciones en lenguajes de programación de bajo nivel como C o C++. Esto haría el software menos legible y más difícil de instalar. En tercer lugar, se ha tratado de evitar trucos de programación, ya que las implementaciones claras son preferibles a las ingeniosas pero indescifrables.

NLTK Taggers

El módulo nltk.tag define varios etiquetadores, que tienen una lista de símbolos (normalmente una oración), asignan una etiqueta a cada símbolo, y devuelven la lista de símbolos etiquetados. La mayoría de los etiquetadores definidos en el módulo nltk.tag

se construyen automáticamente en base a un corpus de entrenamiento. Por ejemplo, el etiquetador "unigram" marca cada palabra w comprobando cual era la etiqueta más frecuente para w que estaba en el corpus de entrenamiento:

```
>>> # Load the brown corpus.
>>> from nltk.corpus import brown
>>> brown_news_tagged = brown.tagged_sents(categories='news')
>>> brown_news_text = brown.sents(categories='news')
>>> tagger = nltk.UnigramTagger(brown_news_tagged[:500])
>>> tagger.tag(brown_news_text[501])
[('Mitchell', 'NP'), ('decried', None), ('the', 'AT'), ('high', 'JJ'),
('rate', 'NN'), ('of', 'IN'), ('unemployment', None), ...]
```

Hay que tener en cuenta que las palabras que el etiquetador no ha visto antes, como "decried", reciben la etiqueta de "None".

En los ejemplos que siguen, se verá el desarrollo de etiquetadores automáticos de partes de la oración basado en el Corpus Brown. Estos son los conjuntos de entrenamiento y de prueba que se usarán:

```
>>> brown_train = brown_news_tagged[100:]
>>> brown_test = brown_news_tagged[:100]
>>> test_sent = nltk.tag.untag(brown_test[0])
```

Estos conjuntos de entrenamiento son pequeños, para que las pruebas se ejecuten más rápido. Para su uso en el mundo real, se utilizarán conjuntos más grandes.

Default Tagger

El etiquetador mas simple es el DefaultTagger, que sólo se aplica la misma etiqueta a todos los símbolos:

```
>>> default_tagger = nltk.DefaultTagger('XYZ')
>>> default_tagger.tag('This is a test'.split())
[('This', 'XYZ'), ('is', 'XYZ'), ('a', 'XYZ'), ('test', 'XYZ')]
```

Dado que 'NN' es la etiqueta más frecuente en el corpus Brown, podemos utilizar un etiquetador que asigna 'NN' a todas las palabras como una línea de base.

```
>>> default_tagger = nltk.DefaultTagger('NN')
>>> default_tagger.tag(test_sent)
[('The', 'NN'), ('Fulton', 'NN'), ('County', 'NN'), ('Grand', 'NN'), ('Jury',
'NN'), ('said', 'NN'), ('Friday', 'NN'), ('an', 'NN'), ('investigation',
'NN'), ('of', 'NN'), ("Atlanta's", 'NN'), ('recent', 'NN'), ('primary',
'NN'), ('election', 'NN'), ('produced', 'NN'), ('``', 'NN'), ('no', 'NN'),
('evidence', 'NN'), ("''", 'NN'), ('that', 'NN'), ('any', 'NN'),
('irregularities', 'NN'), ('took', 'NN'), ('place', 'NN'), ('.', 'NN')]
```

Usando esta línea de base, se consigue una exactitud bastante baja:

```
>>> print 'Accuracy: %4.1f%%' % (
...     100.0 * default_tagger.evaluate(brown_test))
```

```
Accuracy: 14.6%
```

Regexp Tagger

La clase RegexpTagger asigna etiquetas a los símbolos mediante la comparación de sus palabras con una serie de expresiones regulares. El siguiente etiquetador utiliza sufijos de palabras para hacer conjeturas sobre la parte correcta de la etiqueta de la palabra para el Corpus de Brown:

```
>>> regexp_tagger = nltk.RegexpTagger(
...     [(r'^-?[0-9]+(.[0-9]+)?$', 'CD'),   # cardinal numbers
...      (r'(The|the|A|a|An|an)$', 'AT'),   # articles
...      (r'.*able$', 'JJ'),                # adjectives
...      (r'.*ness$', 'NN'),                # nouns formed from adjectives
...      (r'.*ly$', 'RB'),                  # adverbs
...      (r'.*s$', 'NNS'),                  # plural nouns
...      (r'.*ing$', 'VBG'),                # gerunds
...      (r'.*ed$', 'VBD'),                 # past tense verbs
...      (r'.*', 'NN')                      # nouns (default)
... ])
>>> regexp_tagger.tag(test_sent)
[('The', 'AT'), ('Fulton', 'NN'), ('County', 'NN'), ('Grand', 'NN'), ('Jury',
'NN'), ('said', 'NN'), ('Friday', 'NN'), ('an', 'AT'), ('investigation',
'NN'), ('of', 'NN'), ("Atlanta's", 'NNS'), ('recent', 'NN'), ('primary',
'NN'), ('election', 'NN'),('produced', 'VBD'), ('``', 'NN'), ('no', 'NN'),
('evidence', 'NN'), ("''", 'NN'), ('that', 'NN'), ('any', 'NN'),
('irregularities', 'NNS'), ('took', 'NN'), ('place', 'NN'), ('.', 'NN')]
```

Esto da una puntuación más alta que el etiquetador por defecto, pero la precisión es todavía bastante baja:

```
>>> print 'Accuracy: %4.1f%%' % (
...     100.0 * regexp_tagger.evaluate(brown_test))
Accuracy: 33.1%
```

Unigram Tagger

Como se mencionó anteriormente, la clase UnigramTagger encuentra la etiqueta más probable para cada palabra en un corpus de entrenamiento, y luego utiliza esa información para asignar etiquetas a los nuevos símbolos.

```
>>> unigram_tagger = nltk.UnigramTagger(brown_train)
>>> unigram_tagger.tag(test_sent)
[('The', 'AT'), ('Fulton', None), ('County', 'NN-TL'), ('Grand', 'JJ-TL'),
('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR'), ('an', 'AT'),
('investigation', 'NN'), ('of', 'IN'), ("Atlanta's", 'NP$'), ('recent',
'JJ'), ('primary', 'NN'), ('election', 'NN'), ('produced', 'VBD'), ('``',
'``'), ('no', 'AT'), ('evidence', 'NN'), ("''", "''"), ('that', 'CS'),
('any', 'DTI'), ('irregularities', None), ('took', 'VBD'), ('place', 'NN'),
('.', '.')]
```

Esto da una puntuación de exactitud significativamente mayor que el etiquetador por defecto o que el etiquetador de expresiones regulares:

```
>>> print 'Accuracy: %4.1f%%' % (
...     100.0 * unigram_tagger.evaluate(brown_test))
Accuracy: 85.6%
```

Como se mencionó anteriormente, el etiquetador unigram asignará la etiqueta "None" a cualquier palabra que nunca haya visto en los datos de entrenamiento. Se puede evitar este problema dándole al etiquetador unigram un etiquetador auxiliar, que se utilizará cada vez que el etiquetador unigram sea incapaz de elegir una etiqueta:

```
>>> unigram_tagger_2 = nltk.UnigramTagger(brown_train, backoff=regexp_tagger)
>>> print 'Accuracy: %4.1f%%' % (
...     100.0 * unigram_tagger_2.evaluate(brown_test))
Accuracy: 88.2%
```

El uso de un etiquetador auxiliar tiene otra ventaja, porque permite construir un etiquetador unigram más compacto, ya que el etiquetador unigram no necesita almacenar explícitamente las etiquetas para las palabras que el etiquetador auxiliar acertaría en todos los casos. Podemos ver esto usando el método *size(),* que indica el número de palabras para las cuales un etiquetador unigram ha almacenado la etiqueta más probable.

```
>>> print unigram_tagger.size()
14262
>>> print unigram_tagger_2.size()
8722
```

Bigram Tagger

El etiquetador de bigramas es similar al etiquetador de unigramas, la diferencia es que encuentra la etiqueta más probable para cada palabra, según la última etiqueta asignada. (Se llama un etiquetador "bigram" porque utiliza dos tipos de información: La palabra actual y la etiqueta anterior). Durante el entrenamiento, puede consultar la etiqueta inmediatamente anterior. Cuando se ejecuta en los nuevos datos, funciona recorriendo la oración, de izquierda a derecha, y utiliza la etiqueta que acaba de generarse para la palabra anterior.

```
>>> bigram_tagger = nltk.BigramTagger(brown_train,
backoff=unigram_tagger_2)
>>> print bigram_tagger.size()
3379
>>> print 'Accuracy: %4.1f%%' % (
...     100.0 * bigram_tagger.evaluate(brown_test))
Accuracy: 89.6%
```

Trigram Tagger & N-Gram Tagger

Del mismo modo, el etiquetador trigrama encuentra la etiqueta más probable para una palabra, teniendo en cuenta las dos etiquetas anteriores; y el etiquetador de n-gramas encuentra la etiqueta más probable para una palabra, dadas las n-1 etiquetas anteriores. Sin embargo, estos etiquetadores de orden superior sólo son susceptibles de mejorar el rendimiento si hay una gran cantidad de datos de entrenamiento disponibles; de otro

modo, las secuencias que consideran no ocurren con frecuencia suficiente para reunir estadísticas fiables.

```
>>> trigram_tagger = nltk.TrigramTagger(brown_train, backoff=bigram_tagger)
>>> print trigram_tagger.size()
1495
>>> print 'Accuracy: %4.1f%%' % (
...    100.0 * trigram_tagger.evaluate(brown_test))
Accuracy: 89.0%
```

Brill Tagger

El etiquetador de Brill comienza ejecutando un etiquetador inicial, y luego mejora el etiquetado mediante la aplicación de una lista de reglas de transformación. Estas reglas de transformación se aprenden de forma automática desde el corpus de entrenamiento, a partir de uno o más "plantillas de reglas."

Metamap

MetaMap es un programa altamente configurable desarrollado por el Dr. Alan Aronson de la National Library of Medicine (NLM) para asignar texto biomédico al Meta tesauro de UMLS o, de manera equivalente, para descubrir conceptos Meta tesauros mencionadas en el texto. MetaMap utiliza un enfoque intensivo en conocimiento basado en símbolos, procesamiento de lenguaje natural (NLP) y técnicas computacionales lingüísticas. Además de ser aplicado para aplicaciones de minería de datos de IR, MetaMap es uno de las bases del indexador de Texto Médico de la NLM (MTI), que está siendo utilizado además para la indexación semiautomática y totalmente automática de la literatura biomédica en la NLM.

UMLS

El Sistema Unificado de Lenguaje Médico (UMLS) es un compendio de muchos conceptos encontrados en las ciencias biomédicas (creado en 1986). Se proporciona una estructura de mapeo entre estos conceptos y por lo tanto permite traducir entre los diversos sistemas de terminología; también puede ser visto como un tesauro integral y ontología de conceptos biomédicos. UMLS proporciona además instalaciones para el procesamiento del lenguaje natural. Está destinado a ser utilizado principalmente por los desarrolladores de sistemas de informática médica.

UMLS consiste en fuentes de conocimiento (bases de datos) y un conjunto de herramientas de software.

El UMLS fue diseñado y es mantenido por la Biblioteca Nacional de Medicina de EE.UU., se actualiza trimestralmente y puede ser utilizado de forma gratuita. El proyecto fue iniciado en 1986 por Donald A.B. Lindberg, M.D., y actual Director de la Biblioteca de Médica.

El número de recursos disponibles para los investigadores biomédicos es enorme. A menudo esto es un problema debido a la gran cantidad de documentos encontrados cuando se busca en la literatura médica. El propósito de la UMLS es mejorar el acceso a esta literatura, facilitando el desarrollo de sistemas informáticos que entiendan el lenguaje biomédico. Esto se logra mediante la superación de dos obstáculos importantes: la variedad de formas en que los mismos conceptos se expresan en diferentes fuentes legibles por máquina y por diferentes personas y la distribución de información útil entre muchas bases de datos y sistemas dispares.

Propósito del UMLS

El Sistema Unificado de Lenguaje Médico (UMLS) facilita el desarrollo de los sistemas informáticos que se comportan como si "entiendieran" el lenguaje de la biomedicina y la salud. Con ese fin, NLM produce y distribuye las Fuentes de Conocimiento de UMLS (bases de datos) y las herramientas de software asociados (programas). Los desarrolladores utilizan las Fuentes y Herramientas de conocimiento para construir o mejorar los sistemas que crean, procesan, recuperan e integran los datos y la información biomédica y de salud. Las fuentes de conocimiento son de usos múltiples y se utilizan en los sistemas que realizan diversas funciones que implican tipos de

información, como los registros de pacientes, la literatura científica, las directrices y los datos de salud pública. Las herramientas de software asociadas ayudan a los desarrolladores en la personalización o a usar las fuentes de conocimiento UMLS para fines particulares. Las Herramientas Léxicas trabajan más eficazmente en combinación con las fuentes de conocimiento UMLS, pero también se pueden utilizar de forma independiente.

Red Semantica

El propósito de la red semántica es proporcionar una categorización consistente de todos los conceptos representados en el Meta tesauro UMLS y para proporcionar un conjunto de relaciones útiles entre estos conceptos. Toda la información acerca de los conceptos específicos se encuentra en el Meta tesauro. La Red proporciona información sobre el conjunto de tipos semánticos básicos, o categorías, que pueden ser asignados a estos conceptos, y que define el conjunto de relaciones que pueden existir entre los tipos semánticos. La Red Semántica contiene 133 tipos semánticos y 54 relaciones. La Red Semántica sirve como una "autoridad" para los tipos semánticos que se asignan a los conceptos en el Meta tesauro. La red define estos tipos, tanto con descripciones textuales, como por medio de la información inherente a sus jerarquías.

Los tipos semánticos son los nodos de la red, y las relaciones entre ellos son los enlaces. Hay grandes grupos de tipos semánticos para los organismos, las estructuras anatómicas, funciones biológicas, químicas, eventos, objetos físicos y conceptos o ideas. El alcance actual de los tipos semánticos de UMLS es bastante amplio, lo que permite la categorización semántica de una amplia gama de la terminología en varios dominios.

El Meta tesauro consiste en términos desde sus vocabularios de origen. El significado de cada término se define por su origen, de forma explícita, por definición, o anotación; por el contexto (su lugar en la jerarquía); por sinónimos y otras relaciones establecidas entre los términos; y por su uso en la descripción, clasificación o indexación. A cada concepto en el Meta tesauro se le asigna al menos un tipo semántico. En todos los casos, el tipo semántico más específico disponible en la jerarquía se asigna al concepto. Por ejemplo, el concepto de "Macaca" recibe el tipo semántico "mamífero", porque no hay un tipo más específico "Primate" disponible en la red. El nivel de granularidad varía a través de la red. Esto tiene implicaciones importantes para la interpretación del sentido (es decir, el tipo semántico) que se ha asignado a un concepto del Meta tesauro. Por ejemplo, un sub árbol bajo el nodo "objeto físico" es "objeto manufacturado". Sólo tiene dos nodos secundarios, "Producto Sanitario" y "Dispositivo de Investigación". Está claro que hay objetos manufacturados que no son productos sanitarios ni dispositivos de investigación. En lugar de agregar mas tipos semánticos para abarcar múltiples subcategorías adicionales para estos objetos, a los conceptos que no son productos sanitarios ni dispositivos de investigación, simplemente se les asigna el tipo semántico más general "objeto manufacturado".

La siguiente figura muestra una parte de la red. El tipo semántico "Función Biológica" tiene dos hijos, "Función fisiológica" y "Función Patológica", y cada uno de estos a su vez tiene varios hijos y nietos. Cada hijo en la jerarquía está ligado a su padre por el vínculo "es un".

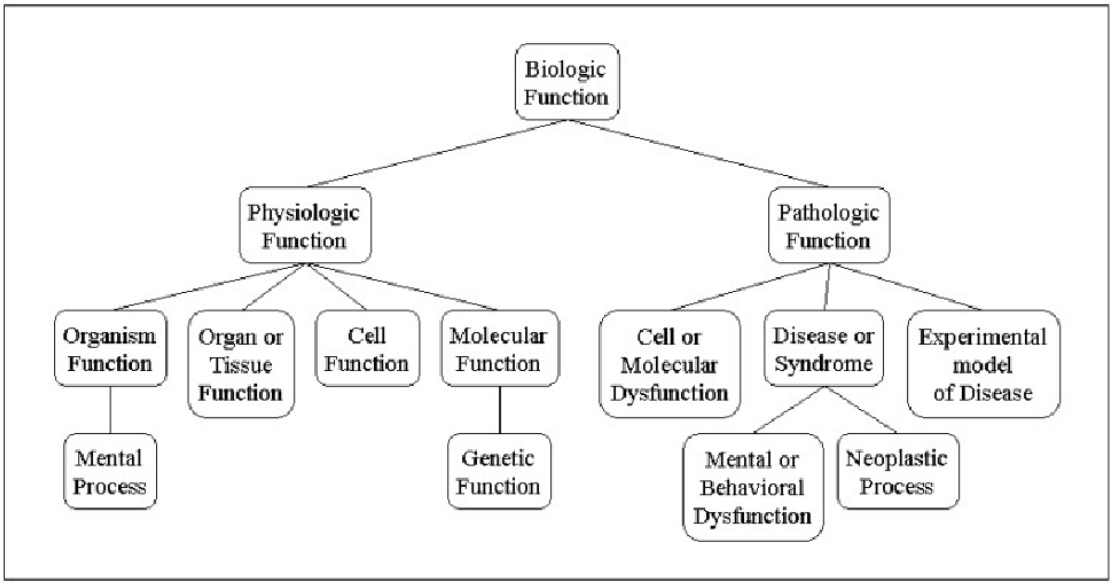

Figura 3 Red semántica

El enlace principal de la Red es el enlace "es un". Esto establece la jerarquía de los tipos dentro de la red y se utiliza para decidir sobre el tipo semántico más específico disponible para su asignación a un concepto en el Meta tesauro. Además, se ha identificado un conjunto de relaciones no jerárquicas entre los tipos. Estos se agrupan en cinco categorías principales, que son en sí mismas las relaciones: "físicamente relacionados con", "espacialmente relacionados con", "temporalmente relacionados con", "funcionalmente relacionados con" y "conceptualmente relacionados con".

La figura muestra una parte de la jerarquía de relaciones de red. La relación "afecta", una de las varias relaciones funcionales, tiene seis hijos, entre ellos "gestiona", "trata" y "impide".

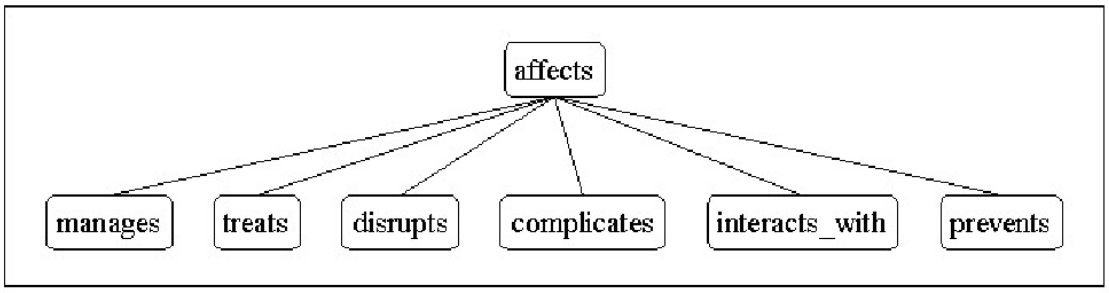

Figura 4 Jerarquía de relaciones

Las relaciones se establecen entre los tipos semánticos de alto nivel en la Red siempre que sea posible, y generalmente se heredan a través del enlace "es un" por todos los hijos de esos tipos. Así, por ejemplo, la relación "proceso de" se establece entre los tipos semánticos "Función Biológica" y "Organismo". Por lo tanto, también se da entre "Función de órganos o tejidos" (que es una "Función fisiológica", que es, a su vez, una "Función Biológica") y "Animal" (que es un "Organismo"). Las relaciones se establecen entre los tipos semánticos y no necesariamente se aplican a todas las instancias de los conceptos que se han asignado a los tipos semánticos. Es decir, la relación puede o no puede mantenerse entre cualquier par de conceptos. Así que, aunque la relación "evaluación de" se mantiene entre el tipo semántico "Signo" y "Atributo Organismo", un signo particular o un atributo en particular no puede estar vinculado por esta relación. Por lo tanto, los signos tales como "sobrepeso" y "fiebre" son evaluaciones de los atributos del organismo "peso corporal " y " temperatura corporal", respectivamente. Sin embargo, "sobrepeso" no es una evaluación de "la temperatura del cuerpo", y la "fiebre" no es una evaluación del "peso corporal".

En algunos casos, habrá un conflicto entre la colocación de los tipos en la Red y en el vínculo que se hereda. Si es así, se dice que la herencia del enlace está bloqueada. Por ejemplo, por herencia, del tipo "Proceso Mental" sería "proceso de" "Planta". Dado que las plantas no son seres sensibles, este enlace se bloquea de forma explícita. En otros casos, la naturaleza de la relación es tal que no debe ser heredado por los hijos de los tipos que se vincula. En ese caso, la relación está definida para los dos tipos semánticos que explícitamente se vinculan, pero bloqueado para todos los hijos de esos tipos. Por ejemplo, "parte conceptual de" conecta "Sistema Corporal" y "Estructura anatómica totalmente formada", pero no debería vincular "Sistema Corporal" para todos los hijos de "Estructura anatómica totalmente formado", como "célula" o "tejido".

Varias partes de la jerarquía MeSH (Medical Subject Headings) han sido etiquetadas con relaciones semánticas de hijo a padre. Todas las secciones de anatomía, enfermedades, psiquiatría y psicología han sido etiquetadas, así como una parte de la sección de ciencias biológicas. Los vínculos que se expresan entre los términos MeSH son, con pocas excepciones, reflejados en la Red Semántica. Es decir, si dos términos MeSH están unidos por una cierta relación, entonces ese vínculo se expresa en la Red como un enlace entre los tipos semánticos que se han asignado a los términos MeSH.

Por ejemplo, "fluido amniótico", que es una "sustancia del cuerpo", es un hijo de "embrión", que es una "estructura embrionaria". La relación marcada entre "fluido amniótico" y su padre "embrión" es "envuelve". Esto es admisible, dado que la relación "sustancia del cuerpo rodea estructura embrionaria" está representada en la red.

La figura muestra una parte de la red semántica, que ilustra las relaciones, ya sean jerárquicas o asociativas, que existen entre los tipos semánticos.

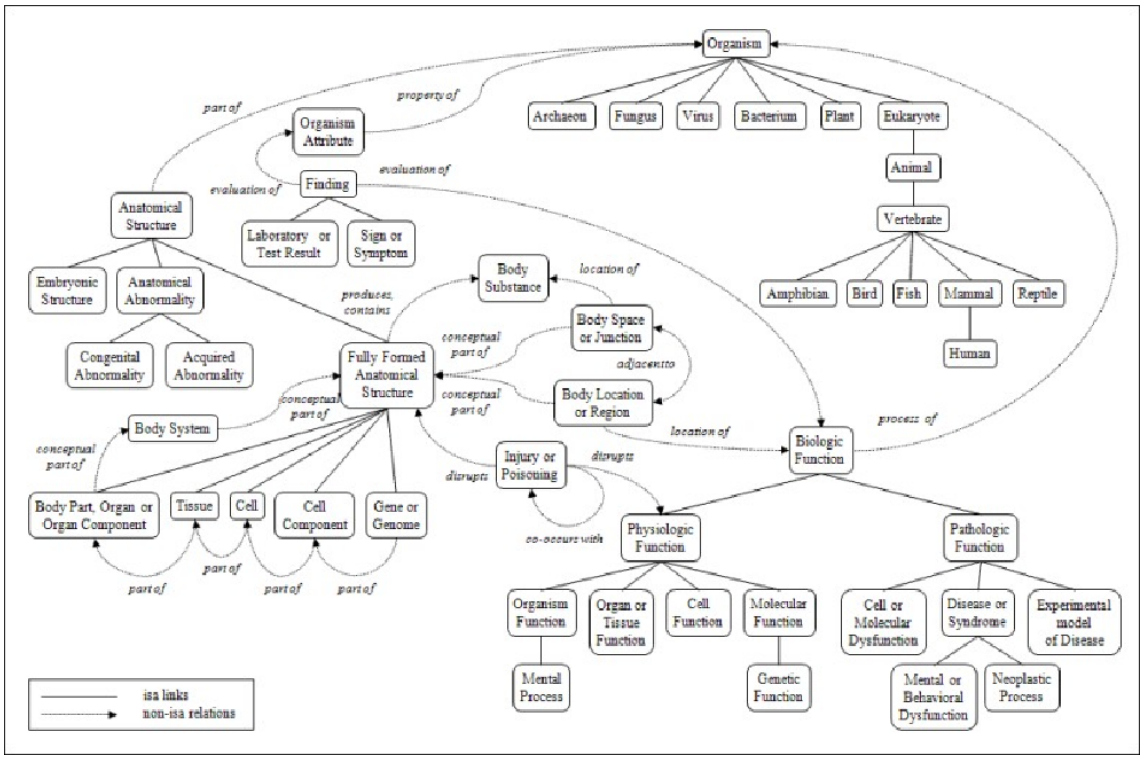

Figura 5 Red semántica y relaciones

Evaluación de Riesgos

Dado que en el contexto de p-medicine que se requiere que todas las partes sean componentes interconectables, se ha optado por realizar un servicio REST, que sea fácilmente reutilizable por diferentes componentes independientemente de la tecnología utilizada.

Un riesgo importante a ser considerado es la capacidad de carga del sistema, por eso se ha decidido realizar un servicio que sea fácilmente escalable, y que no afecte al rendimiento del conjunto. Por ejemplo, si una herramienta requiere trabajar mucho con este servicio, podría fácilmente instalarse un servicio para uso exclusivo de la misma, o podrían instalarse un pool de servidores con el servicio para manejar la carga requerida.

Al momento de realizar el trabajo no se encontraron herramientas similares.

Desarrollo

El desarrollo del trabajo se completó con las siguientes fases: crear un entorno de trabajo, realizar prácticas con las herramientas a utilizar, creación de pruebas para comprobar resultados, creación del servicio web, procesamiento de la información recibida.

Planificación

La planificación fue hecha con la herramienta Trello (www.trello.com), para ello se realizó una descomposición del proyecto, en distintas tareas de complejidad media o baja, necesarias para completarlo.

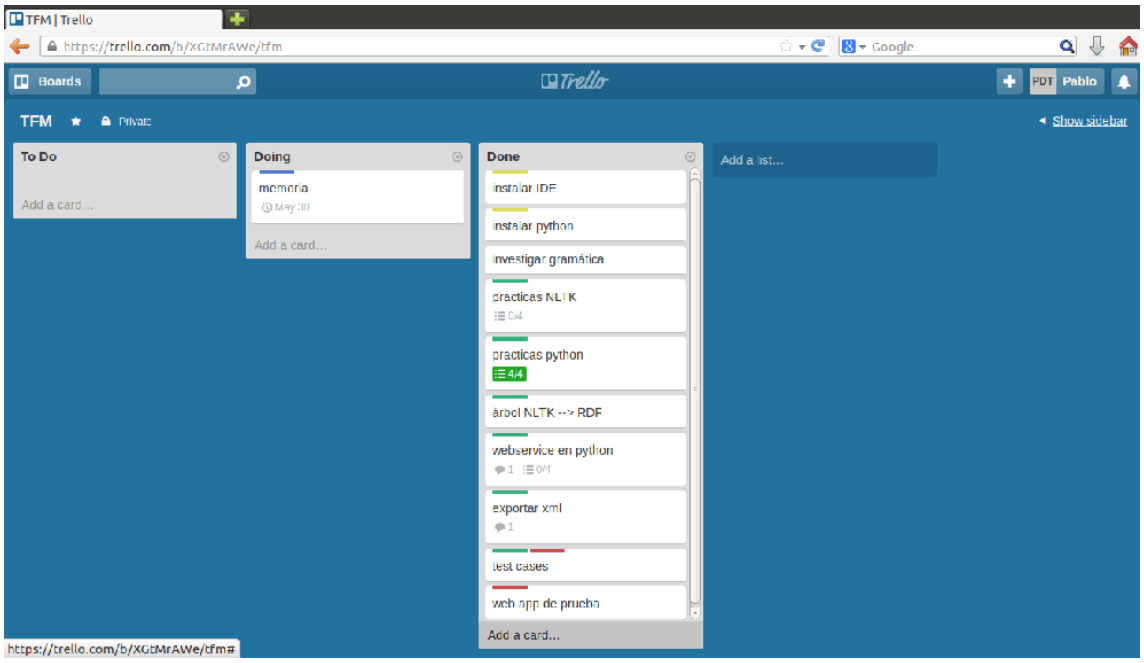

Figura 6 Planificación en Trello.com

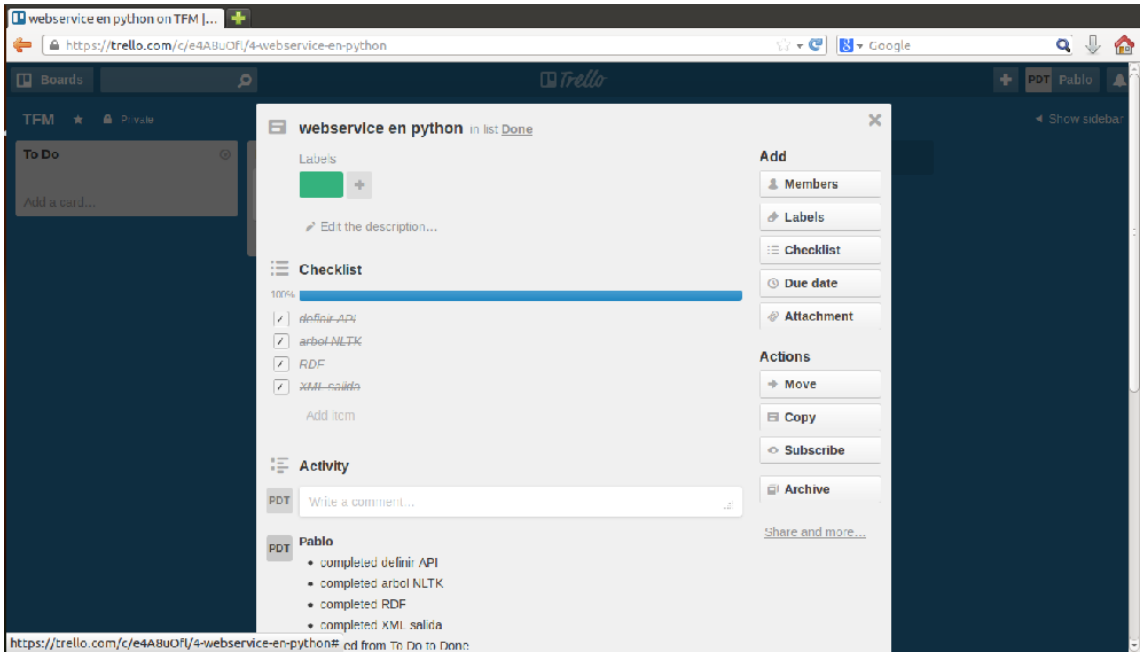

Figura 7 Ejemplo de detalle de tareas

Creación de entorno de trabajo

Para el entorno de trabajo se utilizó un sistema operativo Ubuntu linux, con las siguientes características:

- Ubuntu 12 (precise) 64-bit
- Kernel Linux 3.2.0-59
- Intel Core i3 M-380, 4gb ram

A continuación se indican los comandos necesarios para instalar las herramientas utilizadas:

Instrucciones de Instalación

La forma recomendada para instalar "setuptools" en cualquier sistema es descargar ez_setup.py y ejecutarlo utilizando el entorno de Python. Los diferentes sistemas operativos tienen diferentes técnicas recomendadas para realizar esta rutina, por lo que a continuación indicar los pasos a seguir en un entorno Linux.

Setuptools requiere Python 2.6 o posterior.

Linux (wget)

La mayoría de las distribuciones de Linux ya tienen instalado wget.

Para descargar ez_setup.py y ejecutarlo con Python:

```
> wget https://bootstrap.pypa.io/ez_setup.py -O - | python
```

Es probable que sea necesario llamar al comando usando los privilegios de administrador para instalarlo:

```
> wget https://bootstrap.pypa.io/ez_setup.py -O - | sudo
python
```

Flask Easy to Setup

```
> pip install Flask
```

Instalar NLTK

Instalar Pip:

```
> run sudo easy_install pip
```

Instalar Numpy :

```
> run sudo pip install -U numpy
```

Instalar PyYAML and NLTK:

```
> run sudo pip install -U pyyaml nltk
```

Comprobar la instalación:

```
> run python then type import nltk
```

Instalación de NLTK Data

NLTK viene con muchos corpus, gramáticas de práctica, modelos entrenados, etc.

Para instalar los datos, debe instalar antes primero NLTK, a continuación, se deben realizar los siguientes pasos.

Instalación Interactiva

Ejecutar el intérprete de Python y escribir los comandos:

```
>>> import nltk

>>> nltk.download()
```

Una nueva ventana se abrirá, mostrando el NLTK Downloader. Hacer clic en el menú Archivo y seleccione Cambiar directorio de descarga. Para la instalación central, seleccionar /usr/share/nltk_data. A continuación, seleccionar los paquetes o colecciones que desea descargar.

Prácticas con Python, Flask y NLTK

Antes de comenzar a desarrollar el proyecto, he desarrollado unas prácticas para familiarizarme con los frameworks que he utilizado.

Para ello he desarrollado un servicio REST para administrar una lista de tareas (TODO list), utilizando Flask y Python. El servicio tiene un API, que permite leer, crear y actualizar tareas, utilizando los diferentes verbos de http: GET, POST y PUT respectivamente.

```
@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    return jsonify({ 'tasks': tasks })

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify({ 'task': task[0] })

@app.route('/todo/api/v1.0/tasks', methods = ['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
            abort(400)
    task ={
        'id': tasks[-1]['id'] +1,
        'title': request.json['title'],
        'description': request.json.get('description', ''),
        'done': False
    }
    tasks.append(task)
    return jsonify({'task': task}), 201

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not unicode:
        abort(400)
    if 'done' in request.json and type(request.json["done"]) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
```

Figura 8 Código de práctica Flask y Python

```
ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 262
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Sat, 24 May 2014 13:12:38 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "python tutorial",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$
```

Figura 9 metodo GET del servicio

```
ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Sat, 24 May 2014 13:13:34 GMT

{
  "error": "Not found"
}ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$
```

Figura 10 ver solo una tarea

```
ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":
"Read a book"}' http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 101
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Sat, 24 May 2014 13:14:17 GMT

{
  "task": {
    "description": "",
    "done": false,
    "id": 3,
    "title": "Read a book"
  }
}ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/todo-api$
```

Figura 11 método POST del servicio

La otra práctica realizada, sirvió para familiarizarme con el framework NLTK, consistió en probar las diferentes API y probar distintas configuraciones de etiquetadores para obtener mejores resultados.

```
218  def createUnigramTagger(training, backoff_tagger):
219      unigram_tagger = nltk.UnigramTagger(training, backoff=backoff_tagger)
220      return unigram_tagger
221
222  def createBigramTagger(training, backoff_tagger):
223      bigram_tagger = nltk.BigramTagger(training, backoff=backoff_tagger)
224      return bigram_tagger
225
226  def createTrigramTagger(training, backoff_tagger):
227      trigram_tagger = nltk.TrigramTagger(training, backoff=backoff_tagger)
228      return trigram_tagger
229
230  def createBrillTagger(training, backoff_tagger):
231      templates = [
232          brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,1)),
233          brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (2,2)),
234          brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,2)),
235          brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,3)),
236          brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,1)),
237          brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (2,2)),
238          brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,2)),
239          brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,3)),
240          brill.ProximateTokensTemplate(brill.ProximateTagsRule, (-1, -1), (1,1)),
241          brill.ProximateTokensTemplate(brill.ProximateWordsRule, (-1, -1), (1,1))
242      ]
243
244      trainer = brill.FastBrillTaggerTrainer(backoff_tagger, templates)
245      tagger = trainer.train(training, max_rules=100, min_score=3)
246
247      return tagger
248
```

Figura 12 Prácticas con NLTK

```
    brown_test_set =  brown_news_tagged[:100]

    regexp_tagger = createRegExpTagger()
    unigram_tagger = createUnigramTagger(brown_train, regexp_tagger)
    tagger = createBigramTagger(brown_train, unigram_tagger)
    print 'test_BigramTagger:'
    printAccuracy(tagger, brown_test_set)

def test_TrigramTagger():
    brown_news_tagged = brown.tagged_sents(categories=['news', 'editorial', 'learned', 'reviews', 'government', '
    brown_train = brown_news_tagged[100:]
    brown_test_set =  brown_news_tagged[:100]

    regexp_tagger = createRegExpTagger()
    unigram_tagger = createUnigramTagger(brown_train, regexp_tagger)
    bigram_tagger = createBigramTagger(brown_train, unigram_tagger)
    tagger = createTrigramTagger(brown_train, bigram_tagger)
    print 'test_TrigramTagger:'
    printAccuracy(tagger, brown_test_set)


def test_BrillTagger():
    brown_news_tagged = brown.tagged_sents(categories=['news', 'editorial', 'learned', 'reviews', 'government', '
    brown_train = brown_news_tagged[100:]
    brown_test_set =  brown_news_tagged[:100]

    regexp_tagger = createRegExpTagger()
    unigram_tagger = createUnigramTagger(brown_train, regexp_tagger)
    bigram_tagger = createBigramTagger(brown_train, unigram_tagger)
    trigram_tagger = createTrigramTagger(brown_train, bigram_tagger)
    tagger = createBrillTagger(brown_train, trigram_tagger)
    print 'test_BrillTagger:'
    printAccuracy(tagger, brown_test_set)
```

Figura 13 Prácticas con NLTK

En la siguiente figura puede verse como mejora la precisión de los distintos etiquetadores:

```
test_RegExpTagger:
Accuracy: 19.0%

test_unigramTagger:
Accuracy: 89.9%

test_BigramTagger:
Accuracy: 91.4%

test_TrigramTagger:
Accuracy: 91.3%

test_BrillTagger:
Accuracy: 92.7%
FIN
ptoloza@Ub-VirtualBox:~/Code/TFM/practicas/nltk$
```

Figura 14 Precisión de los etiquetadores

Vista de Componentes

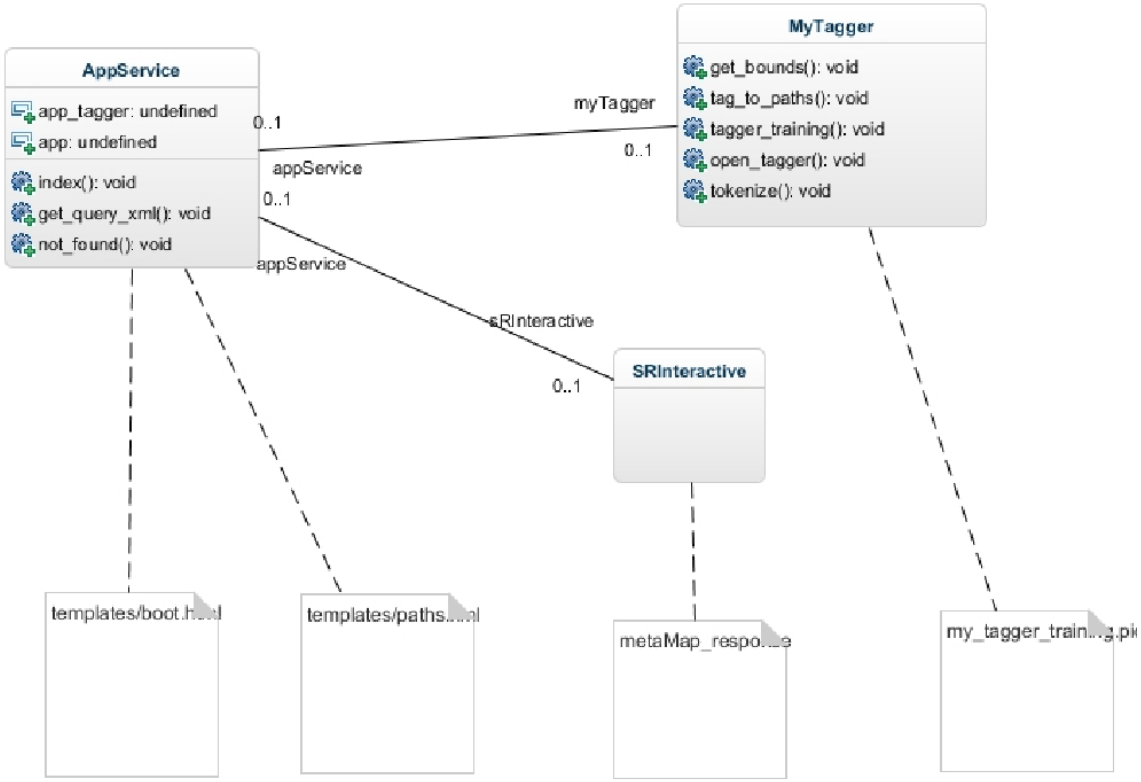

Figura 15 Componentes

- AppService tienen la implementación del servicio rest, administra los request y valida los parámetros recibidos
- My_tagger crea y entrena el etiquedor.
- my_tagger_trainning.pickle archivo que almacena el etiquetador ya entrenado para mejorar el desempeño.
- En la carpeta “static” está los archivos javascript que se utiliza en la interfaz de pruebas.
- En la carpeta “templates” está las plantilla utilizadas, una es la interfaz de pruebas y la otra es la utilizada para generar la respuesta XML.
- SRInteractive es el programa utilizado para interactuar con el servicio web de metamap.

Creación del servicio Web

El servicio REST está implementado en el archivo "appService.py", aqui se invocan las librerías necesarias, se crea el "tagger" de NLTK y se configuran las url a las cuales responderá el servicio. Estas urls son:

La raiz del servicio ("/"): es la url por default, y mostrará la interfaz de pruebas del servicio. Esta interfaz está localizada en el directorio "templates". Solo responderá a peticiones del tipo "GET".

```
12
13
14  @app.route('/')
15  def index():
16      return render_template("boot.html")
17
```

Figura 16 Servicio de interfaz de pruebas

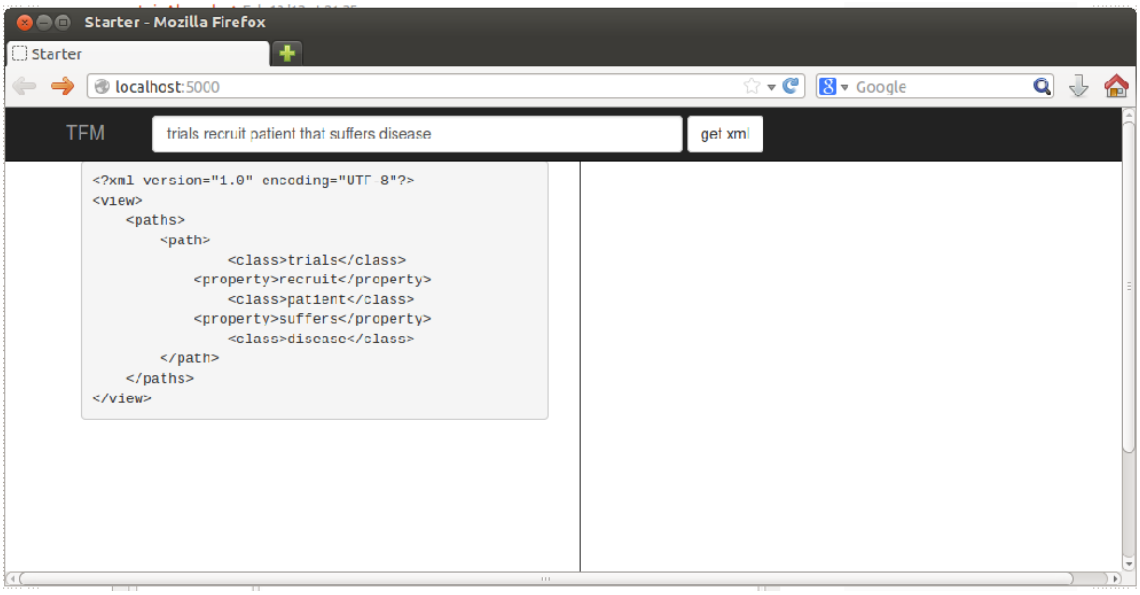

Figura 17 Interfaz de pruebas

La siguiente url a la que responderá el servicio es "/service/api/v1.0/query.xml", que es la url donde se realizan las queries, se atenderá peticiones "GET" y "POST". La peticiones deben incluir en un parámetro llamado "query", las consultas a realizar. Las consultas son enviadas al "tagger" de NLTK para procesarlas, y con los resultados obtenidos se genera la respuesta XML con el formato correspondiente. En caso de no recibir ningún parámetro o recibirlo vacío, el servicio genera un error indicando lo sucedido.

```
19  @app.route('/service/api/v1.0/query.xml', methods = ['GET','POST'])
20  def get_query_xml():
21      query = request.values.get('query', '')
22
23      tokens = my_taggers.tokenize(query)
24      # tagger = my_taggers.open_tagger()
25      tagger = app_tagger
26      tags = tagger.tag(tokens)
27      paths = my_taggers.tag_to_paths(tags)
28      internal_bounds = my_taggers.get_bounds(tags)
29
30      paths = render_template("paths.xml", paths = paths, bounds = internal_bounds)
31      response = make_response(paths)
32      response.headers["Content-Type"] = "application/xml"
33      return response
34
```

Figura 18 Urls aceptadas

Para cualquier otra url que reciba el servicio, se devolverá un error 404, indicando que no se encuentra el recurso solicitado.

```
35
36
37  @app.errorhandler(404)
38  def not_found(error):
39      return make_response(jsonify({'error': 'Not found'}), 404)
40
41  #def index():
```

Figura 19 recursos no encontrados

Procesamiento de Información

El entrenamiento del etiquetador o "tagger", consiste en dos etapas principales: la selección de un corpus para el entrenamiento y la selección y configuración de la cadena de etiquetadores.

Para el desarrollo de este proyecto he decidido utilizar el corpus BROWN, que contiene más de un millón de palabras, en textos de diferentes temas. Los temas que he seleccionado para el entrenamiento son: 'news', 'editorial', 'learned' y 'reviews', ya que son lo que más se aproximan al tipo de información sobre el que se trabajará. De todos modos, es fácilmente configurable y puede cambiarse en cualquier momento.

Para la selección de la cadena de etiquetadores, se ha optado por la siguiente configuración: Fast Brill Tagger Trainer, Trigram tagger, Bigram Tagger, Unigram Tagger, Regexp Tagger, dado que la relación entre la precisión y la velocidad es adecuada para el caso.

```
80  def tagger_training():
81      brown_news_tagged = brown.tagged_sents(categories=['news', 'editorial', 'learned', 'reviews'])
82      brown_train = brown_news_tagged[100:]
83
84      regexp_tagger = nltk.RegexpTagger(
85       [
86          (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),
87          (r'.*ould$', 'MD'),
88          (r'.*ing$', 'VBG'),
89          (r'.*ed$', 'VBD'),
90          (r'.*ness$', 'NN'),
91          (r'.*ment$', 'NN'),
92          (r'.*ful$', 'JJ'),
93          (r'.*ious$', 'JJ'),
94          (r'.*ble$', 'JJ'),
95          (r'.*ic$', 'JJ'),
96          (r'.*ive$', 'JJ'),
97          (r'.*ic$', 'JJ'),
98          (r'.*est$', 'JJ'),
99          (r'^a$', 'PREP'),
100         (r'.*', 'NN')
101
102     ])
103
104     unigram_tagger_2 = nltk.UnigramTagger(brown_train, backoff=regexp_tagger)
105     bigram_tagger = nltk.BigramTagger(brown_train, backoff=unigram_tagger_2)
106     trigram_tagger = nltk.TrigramTagger(brown_train, backoff=bigram_tagger)
107
108     templates = [
109         brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,1)),
110         brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (2,2)),
111         brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,2)),
112         brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule, (1,3)),
113         brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,1)),
114         brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (2,2)),
115         brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,2)),
116         brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule, (1,3)),
117         brill.ProximateTokensTemplate(brill.ProximateTagsRule, (-1, -1), (1,1)),
118         brill.ProximateTokensTemplate(brill.ProximateWordsRule, (-1, -1), (1,1))
119     ]
120
121     # trainer = brill.FastBrillTaggerTrainer(raubt_tagger, templates)
122     trainer = brill.FastBrillTaggerTrainer(trigram_tagger, templates)
123     braubt_tagger = trainer.train(train_sents, max_rules=100, min_score=3)
124
125
126     # return trigram_tagger
127     return braubt_tagger
128
```

Figura 20 Entrenamiento del etiquetador

Para optimizar el funcionamiento, el etiquetador se almacena en un fichero, en caso de querer modificar la configuración del etiquetador, basta con borrar el fichero y reiniciar el servicio para que vuelva a entrenarse.

```
def open_tagger():
    # print 'open_tagger'

    training_file = 'my_tagger_training.pickle'
    if not os.path.isfile(training_file):
        f = open(training_file,'w')
        tagger = tagger_training()
        pickle.dump(tagger,f)
        f.close()
    else:
        #And then when you need to load the tagger you use:
        # print 'FILE FOUNDED...'
        f = open(training_file,'r')
        tagger = pickle.load(f)
        f.close()

    return tagger
```

Figura 21 Almacenamiento del etiquetador

Utilizando el servicio web de metamap, se obtienen las entidades mas relevantes del texto y se marcan para luego ser procesadas por el etiquetador. Para ello se utiliza un programa escrito en java que utiliza el api web de matamap.

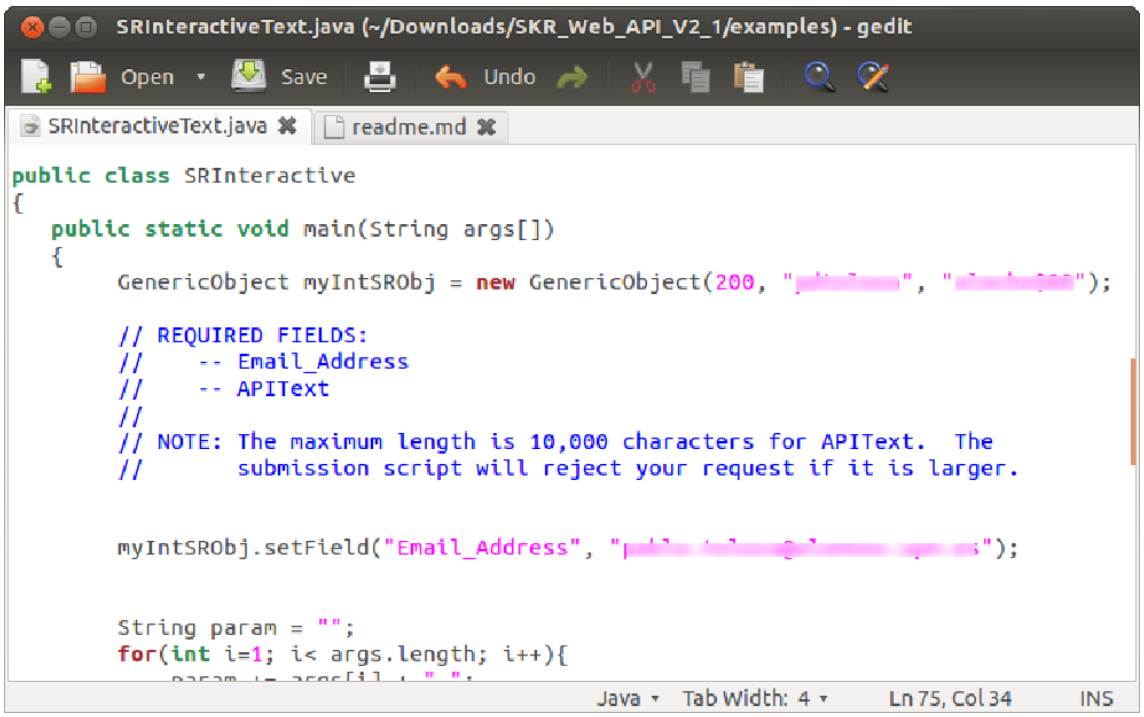

Figura 22 Preparación de parametros Metamap

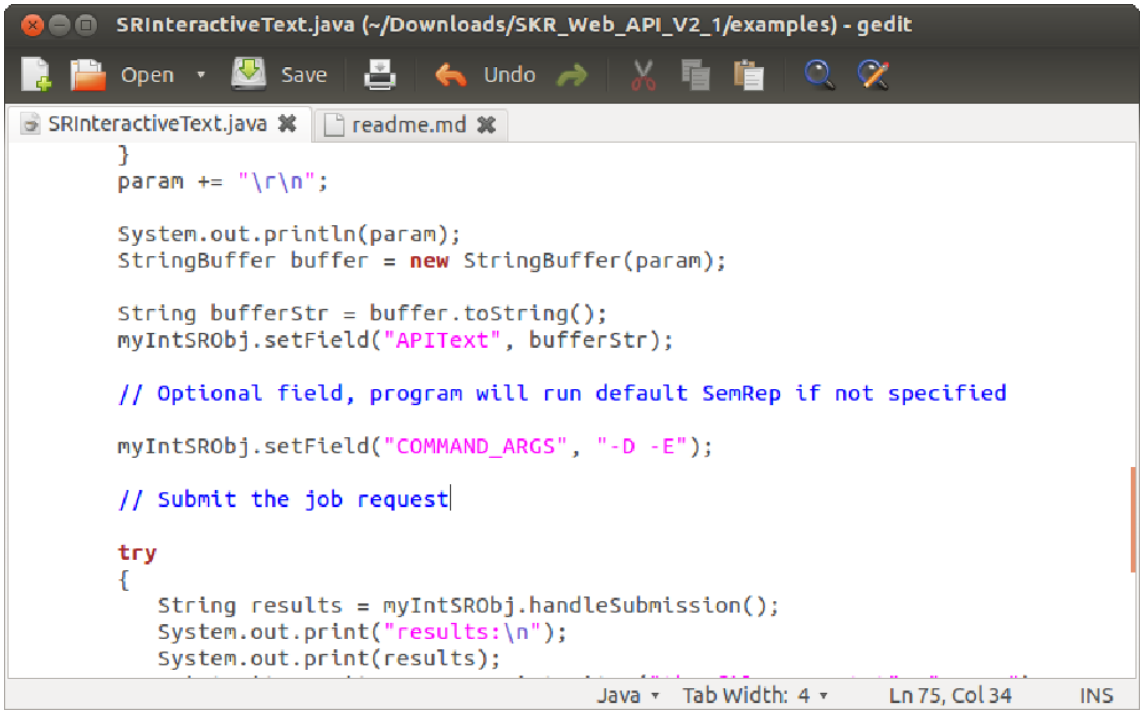

Figura 23 Llamada al servicio Metamap

Luego de etiquetar las diferentes partes de la oración, se analizan las relaciones entre las mismas y se genera una lista con las relaciones existentes entre las distintas partes de la oración. Esta información es necesaria para la generación del XML que se genera como respuesta.

El formato del xml que se genera como respuesta, tiene una especificación concreta. Este XML usa un tag raíz <view>, del que cuelga un tag <paths>. De este tag cuelgan uno o más tags <path>. Cada tag <path> representa un path lineal de nodos y arcos del grafo. Sus sub-tags <class> y <property> sirven para indicar los valores de los nodos y arcos de dicho path. Siempre se debe empezar y terminar por un tag <class>.

Para indicar la unión de dos paths en un nodo concreto se utiliza el atributo "internalBound" en los tags <class> de los dos paths que se quieran unir. Al tener las dos clases el mismo valor para dicho atributo, se especifica que esas dos clases son el mismo nodo en el grafo. Da igual qué valor tenga el atributo "internalBound", mientras que sea igual en las dos clases que se quieren unir. Es importante que las clases que están enlazadas mediante un internal bound tengan exactamente el mismo nombre.

Para ello utilizan el siguiente template:

```
<?xml version="1.0" encoding="UTF-8"?>
<view>
    <paths>
    {%- for path in paths %}
        <path>
        {%- for w in path -%}
            {%- if w[1].startswith("N") %}
                {%- if bounds[w[0]]|length >0 %}
            <class internalBound="{{ bounds[w[0]] }}">{{ w[0] }}</class>
                {%- else %}
            <class>{{ w[0] }}</class>
                {%- endif %}

            {%- else %}
            <property>{{ w[0] }}</property>
            {%- endif %}
        {%- endfor %}
        </path>
    {%- endfor %}
    </paths>
</view>

```

Figura 24 Template para la generación de XML

los parámetros que recibe son los paths de la query las relaciones internas.

Resultados

Como se resultado del trabajo se ha desarrollado un servicio REST con las siguientes funcionalidades:

- Interpretar lenguaje natural
- Identificar entidades de las ciencias médicas
- Generar grafos RDF

Los resultados obtenidos son satisfactorios, ya que uno de los principales objetivos era la identificación de las entidades y esto se consigue gracias METAMAP, el desempeño obtenido es suficiente, pero también puede ser mejorable. Por ejemplo:

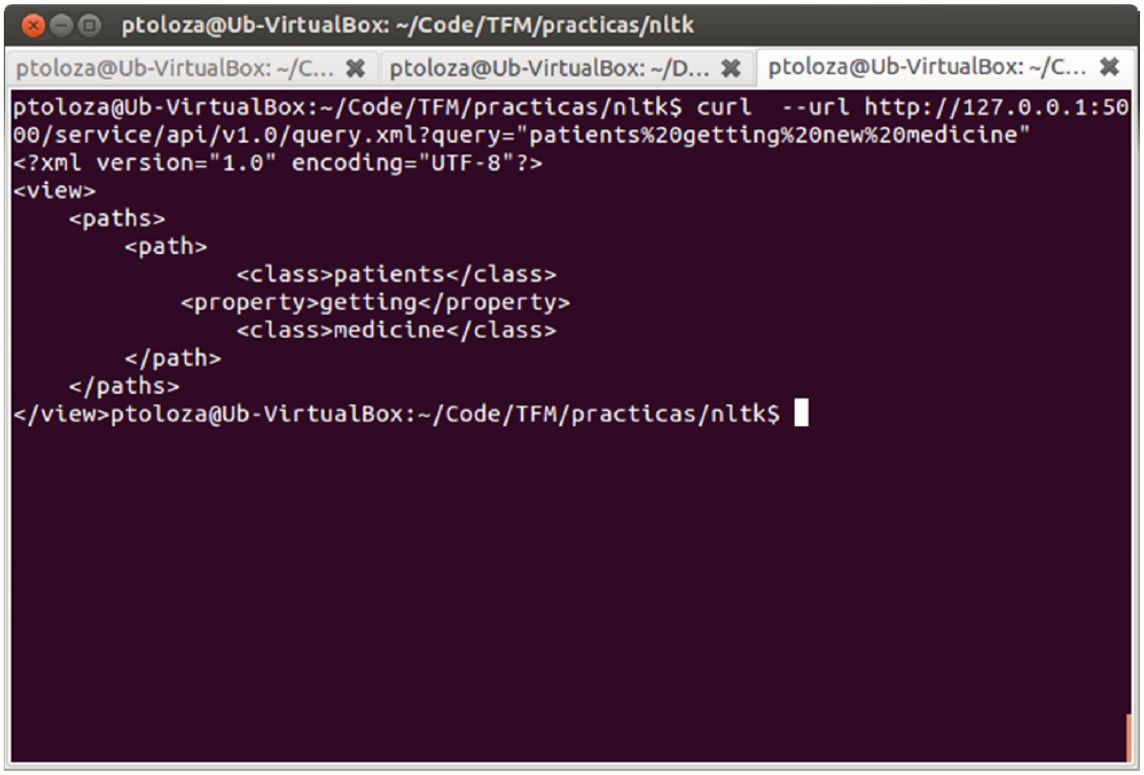

Figura 25 Ejemplo de llamada al servicio - Metodo GET

En la imagen anterior puede verse la respuesta del servicio cuando se le envía la consulta = "patients getting new medicine". El resultado es el esperado, dado que lo etiquetadores han identificado correctamente las palabras en la oración.

Pero cuando lo etiquetadores fallan al identificar una palabra, el resultado no es el esperado:

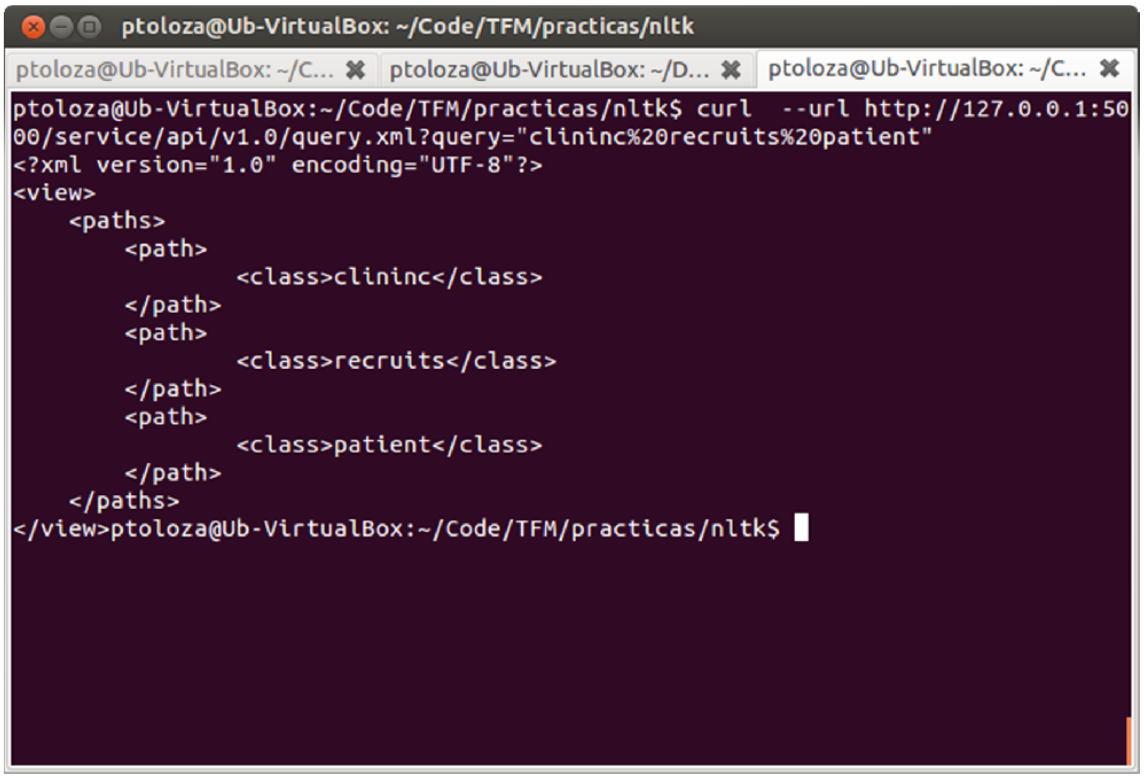

Figura 26 Fallo de los etiquetadores

En este caso lo etiquetadores no han podido identificar correctamente al verbo “recruits” (en español: reclutan), sino que lo confunden con el sustantivo (en español: reclutas). Este tipo de errores si puede mitigar mejorando el entrenamiento de los etiquetadores.

Dado que es una aplicación para un contexto específico, y no es de uso genérico, al cumplir los requerimientos propuestos, se están satisfaciendo las necesidades del cliente.

La aplicación es fácilmente escalable, solo bastaría con instalar el servicio en varios servidores, y utilizando balanceadores de carga, podría soportar cualquier carga de trabajo. Un punto importante dentro de la infraestructura de TI de p-medicine.

Otro punto importante es que el servicio utiliza solo estándares, como JSON y XML, lo que permite que pueda ser utilizado por cualquier aplicación o sistema independientemente de las tecnologías con que éstos estén desarrollados.

Ver anexo “Código Fuente Proyecto”.

Conclusiones

Objetivos Cumplidos:

- Desarrollar Servicio REST capaz de interpretar lenguaje natural.
- Generar XML con formato definido para generar grafos RDF.
- Servicio modular y escalable.

Como se menciona anteriormente, el trabajo no es definitivo y puede ser mejorable en los siguientes aspectos:

El desempeño, si bien se han utilizado técnicas para disminuir el tiempo utilizado por los etiquetadores para poder procesar las consultas, el sistema todavía demora un poco dado que el servidor de Metamap está remoto y las peticiones no son manejadas en forma inmediata, esto podría mejorarse instalando un servidor propio de Metamap en el que las peticiones se ejecuten inmediatamente.

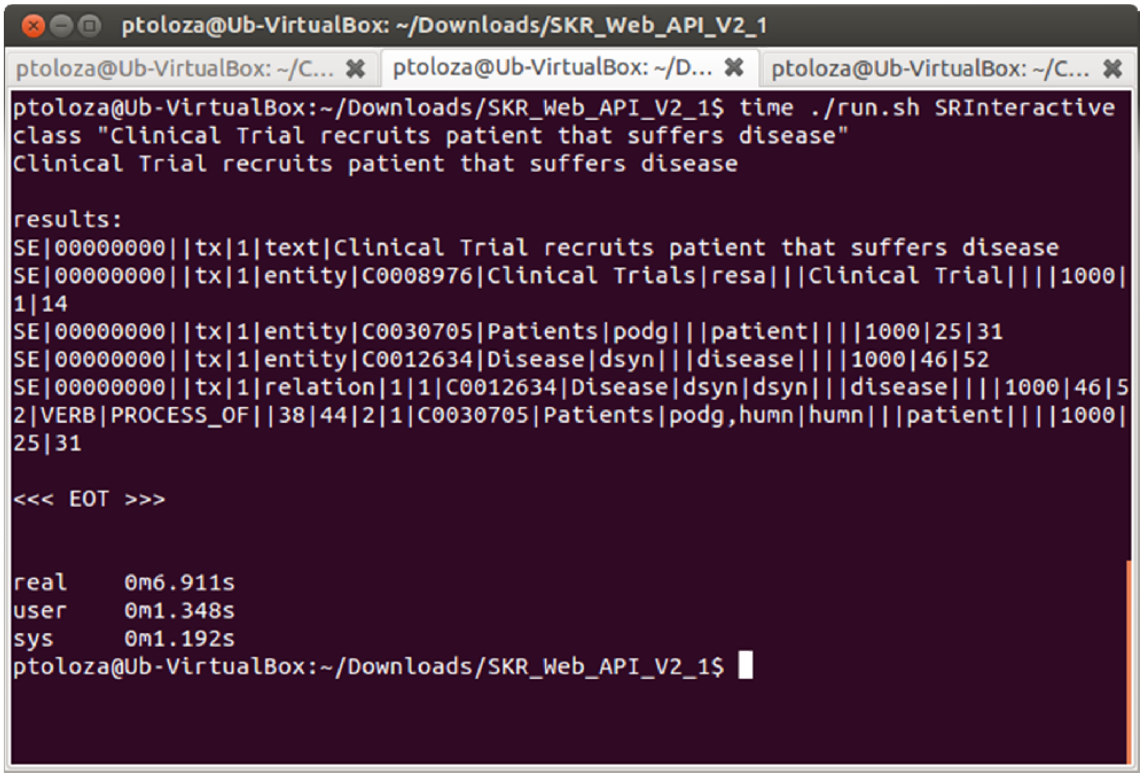

Figura 27 Tiempo de respuesta de Metamap

En la imagen anterior se observa que el tiempo de respuesta de Metamap es de aproximadamente 6 segundos, esto es porque el servicio no procesa las peticiones inmediatamente.

Los etiquetadores pueden mejorar su desempeño si son entrenados con un corpus de mayor tamaño y con contenido más cercano al contexto del proyecto. Si bien el corpus de Brown, que se ha utilizado en el desarrollo de este trabajo, fue suficiente y presentó buenos resultados, en la actualidad existen corpus que son más específicos y con mayor contenido, pudiendo conseguir mayor calidad en los etiquetados.

En cuando al lenguaje de programación, Python ha resultado ser sencillo de aprender y bastante intuitivo, su desempeño en cuanto al procesamiento de datos es aceptable, dado que las consultas que se esperan no son de gran tamaño. Lo mismo se puede decir acerca de Flask, su facilidad de configuración y características lo hacen ideal para este tipo de desarrollos.

Líneas futuras

Como líneas futuras, el servicio podría dar soporte a otros estándares, como por ejemplo usar schemas para los xml, o dar respuestas en JSON, en texto plano o en un fichero.

Dada la modularidad del servicio, no presenta muchas limitaciones en cuanto a tecnología de hardware, las principales limitaciones que presenta es la precisión de los etiquetadores de las partes de la oración, pero esto puede mejorarse utilizando un corpus de mayor extensión y cuyos textos estén más relacionados con la medicina y las ciencias.

Otra de las limitaciones es el uso que Metamap en forma remota, ya que los tiempos de respuesta son algo lentos, pero puede ser fácilmente solucionado, instalando Metamap en un servidor propio.

Bibiliografía

Steven Bird; Ewan Klein; Edward Loper, (Junio 2009): Natural Language Processing with Python (http://shop.oreilly.com/product/9780596516499.do)

Fielding, Roy T.; Taylor, Richard N. (Mayo 2002), "Principled Design of the Modern Web Architecture"

Fielding, Roy Thomas (2000), Architectural Styles and the Design of Network-based Software Architectures

Mark Lutz (Junio 2013) “Learning Python, 5th Edition” (http://shop.oreilly.com/product/0636920028154.do)

UMLS Reference Manual (http://www.ncbi.nlm.nih.gov/books/NBK9681/)

Stream Hacker (http://streamhacker.com/)

http://flask.pocoo.org/docs/api/

Anexos

Código Fuente Proyecto

Service/appService.py

```
#!flask/bin/python
from flask import Flask, jsonify
from flask import render_template
from flask import abort
from flask import make_response
from flask import request
import my_taggers


app = Flask(__name__)
app_tagger = my_taggers.open_tagger()


@app.route('/')
def index():
	return render_template("boot.html")


@app.route('/service/api/v1.0/query.xml', methods = ['GET','POST'])
def get_query_xml():
	query = request.values.get('query', '')
	tokens = my_taggers.tokenize(query)
	# tagger = my_taggers.open_tagger()
	tagger = app_tagger
	tags = tagger.tag(tokens)
	paths = my_taggers.tag_to_paths(tags)
	internal_bounds = my_taggers.get_bounds(tags)
	paths = render_template("paths.xml", paths = paths, bounds =
internal_bounds)
	response = make_response(paths)
	response.headers["Content-Type"] = "application/xml"
	return response


@app.errorhandler(404)
def not_found(error):
	return make_response(jsonify({'error': 'Not found'}), 404)


if __name__ == '__main__':
	app.run(debug = True)
```

Service/my_taggers.py

```
#! /usr/bin/env python
import nltk
from nltk.corpus import brown
import sys
import pickle
import os.path
from nltk.tag import brill


def printAccuracy(tagger, test):
	print 'Accuracy: %4.1f%%' % (
		100.0 * tagger.evaluate(test))


def get_bounds(tags):
	bounds = {}
	i = 1


	for w in tags:
		if w[1].startswith('N'):
			if w[0] in bounds:
				bounds[w[0]] += 1
			else:
				bounds[w[0]] = 0


	words_with_bounds = [w for w in bounds.keys() if bounds[w] > 0]
	internal_bounds = {}
	for w in words_with_bounds:
		internal_bounds[w] = 'InternalBound{0}'.format(i)
		i += 1
	# print internal_bounds
	return internal_bounds


def tag_to_paths(tags):
	# print tags
	# print type(tags)
	# print type(tags[0])
	paths = []
	last = ''
	path = []


	for w in tags:
		if w[1].startswith("N"):
			if last == 'c':
				paths.append(path)
```

```
                        path = []
                    path.append(w)
                    last = 'c'
            if w[1].startswith("V") or w[1].startswith("HV")or
w[1].startswith("BE") :
                    path.append(w)
                    last = 'p'
      paths.append(path)


      return paths


def tagger_training():
      brown_news_tagged = brown.tagged_sents(categories=['news',
'editorial', 'learned', 'reviews'])
      brown_train = brown_news_tagged[100:]
      regexp_tagger = nltk.RegexpTagger(
     [
          (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),
            (r'.*ould$', 'MD'),
          (r'.*ing$', 'VBG'),
          (r'.*ed$', 'VBD'),
          (r'.*ness$', 'NN'),
          (r'.*ment$', 'NN'),
          (r'.*ful$', 'JJ'),
          (r'.*ious$', 'JJ'),
          (r'.*ble$', 'JJ'),
          (r'.*ic$', 'JJ'),
          (r'.*ive$', 'JJ'),
          (r'.*ic$', 'JJ'),
          (r'.*est$', 'JJ'),
          (r'^a$', 'PREP'),
          (r'.*', 'NN')
      ])


      unigram_tagger_2 = nltk.UnigramTagger(brown_train,
backoff=regexp_tagger)
      bigram_tagger = nltk.BigramTagger(brown_train,
backoff=unigram_tagger_2)
      trigram_tagger = nltk.TrigramTagger(brown_train,
backoff=bigram_tagger)



      templates = [

brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule,
(1,1)),
```

```
brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule,
(2,2)),

brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule,
(1,2)),

brill.SymmetricProximateTokensTemplate(brill.ProximateTagsRule,
(1,3)),

brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule,
(1,1)),

brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule,
(2,2)),

brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule,
(1,2)),

brill.SymmetricProximateTokensTemplate(brill.ProximateWordsRule,
(1,3)),
          brill.ProximateTokensTemplate(brill.ProximateTagsRule, (-1,
-1), (1,1)),
          brill.ProximateTokensTemplate(brill.ProximateWordsRule, (-1,
-1), (1,1))
      ]


      trainer = brill.FastBrillTaggerTrainer(trigram_tagger,
templates)
      braubt_tagger = trainer.train(train_sents, max_rules=100,
min_score=3)


      return braubt_tagger


def open_tagger():
      # print 'open_tagger'
      training_file = 'my_tagger_training.pickle'
      if not os.path.isfile(training_file):
            f = open(training_file,'w')
            tagger = tagger_training()
            pickle.dump(tagger,f)
            f.close()
      else:
            #And then when you need to load the tagger you use:
            # print 'FILE FOUNDED...'
            f = open(training_file,'r')
```

```
            tagger = pickle.load(f)
            f.close()
      return tagger


def tokenize(sentence):
      return nltk.word_tokenize(sentence)


if __name__ == "__main__":
      test_sent1 = """
            Patients that receive chemotherapy and are female
      """
      test_sent1 = test_sent1.split()


      test_sent2 = """
            Clinical Trial recruits patient that suffers disease
      """
      test_sent2 = test_sent2.split()


      test_sent4 = test_sent3 = """
            Patient undergoes a biopsy, which reveals a sample that
has size. The biopsy happens before chemotherapy
      """
      test_sent3 = test_sent3.split()
      tokens = nltk.word_tokenize(test_sent4)


      tagger = open_tagger()
      tags = tagger.tag(test_sent1)
      print tags
      print '---------------'


      tags = tagger.tag(test_sent2)
      print tags
      tags = tagger.tag(tokens)
      print '---------------'


      paths = tag_to_paths(tags)
      print tags
      print paths
      get_bounds(tags)


      print "FIN"
```

Service/templates/boot.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="shortcut icon" href="../../assets/ico/favicon.ico">


    <title>Starter</title>


    <!-- Bootstrap core CSS -->
    <!-- <link href="../../dist/css/bootstrap.min.css"
rel="stylesheet"> -->
    <link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"
>


    <!-- Custom styles for this template -->
    <link href="starter-template.css" rel="stylesheet">
    <style>
      body {
        padding-top: 50px;
      }


      .starter-template {
        padding: 40px 15px;
        text-align: center;
      }
    </style>


    <!-- Just for debugging purposes. Don't actually copy this line! -
->
    <!--[if lt IE 9]><script src="../../assets/js/ie8-responsive-file-
warning.js"></script><![endif]-->
    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and
media queries -->
    <!--[if lt IE 9]>
      <script
src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></scrip
t>
      <script
```

```
src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></sc
ript>
    <![endif]-->
  </head>


  <body>
    <div class="navbar navbar-inverse navbar-fixed-top"
role="navigation">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target=".navbar-collapse">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" href="#">TFM</a>
        </div>
        <div class="collapse navbar-collapse">
          <div class="col-md-9">


          <form class="navbar-form navbar-left " role="search">


            <div class="form-group">
              <input type="text" class="form-control "
placeholder="Search" id="query" style="width: 500px">
            </div>
            <!-- <button type="button" class="btn btn-default"
id="submit">Submit</button> -->
            <button type="button" class="btn btn-default"
id="submitxml">get xml</button>
          </form>


          </div>
      </div>


    </div>


    <div class="container">
      <div class="row-fluid">
        <div class="col-md-6">
          <pre id="result">
          </pre>
```

```
        </div>
        <div class="col-md-6">
          <div id="chart" style="float: left;border-width: 1px;
border-style: solid;width:100%;min-height:500px;height:100%">
          </div>


          <div style="border: 1px solid black; background:
white;display:none;position: absolute;" id="literals">
            <h3 style="padding:5px;" id="literalsubject"></h3>
            <div style="padding:5px;" id="literalmsg"></div>
            <table class="table table-hover" id="literaltable">
              <thead>
                <tr>
                  <th>Property</th><th>Value</th>
                </tr>
              </thead>
              <tbody id="literalbody">
              </tbody>
            </table>
          </div>
        </div>
      </div>


    </div><!-- /.container -->


    <!-- Bootstrap core JavaScript
    ================================================== -->
    <!-- Placed at the end of the document so the pages load faster --
>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js
"></script>
    <!-- <script src="../../dist/js/bootstrap.min.js"></script> -->
  <script
src="//netdna.bootstrapcdn.com/bootstrap/3.1.1/js/bootstrap.min.js"></
script>
  <script src="http://d3js.org/d3.v3.min.js" charset="utf-8"></script>
  <script src="/static/main.js"></script>



  </body>
</html>
```

Service/templates/paths.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<view>
	<paths>
	{%- for path in paths %}
		<path>


		{%- for w in path -%}
			{%- if w[1].startswith("N") %}
				{%- if bounds[w[0]]|length >0 %}
			<class internalBound="{{ bounds[w[0]] }}">{{ w[0]
}}</class>
				{%- else %}
			<class>{{ w[0] }}</class>
				{%- endif %}


			{%- else %}
			<property>{{ w[0] }}</property>
			{%- endif %}


		{%- endfor %}
		</path>
	{%- endfor %}
	</paths>
</view>
```

SRInteractive.java

```
/*
===========================================================================
*
*                            PUBLIC DOMAIN NOTICE
*               National Center for Biotechnology Information
*         Lister Hill National Center for Biomedical Communications
*
*  This software is a "United States Government Work" under the terms
of the
*  United States Copyright Act.  It was written as part of the
authors' official
*  duties as a United States Government contractor and thus cannot be
*  copyrighted.  This software is freely available to the public for
use. The
*  National Library of Medicine and the U.S. Government have not
placed any
*  restriction on its use or reproduction.
*
*  Although all reasonable efforts have been taken to ensure the
accuracy
*  and reliability of the software and data, the NLM and the U.S.
*  Government do not and cannot warrant the performance or results
that
*  may be obtained by using this software or data. The NLM and the
U.S.
*  Government disclaim all warranties, express or implied, including
*  warranties of performance, merchantability or fitness for any
particular
*  purpose.
*
*  Please cite the authors in any work or product based on this
material.
*
===========================================================================
```

```
*/


/**
 * Example program for submitting an Interactive SemRep request.
 *
 * This example shows how to setup a basic Interactive SemRep request.
 * This runs the latest version of SemRep with Full Fielded Output (-
D).
 *
 * @author  Jim Mork
 * @version 1.0, June 16, 2011
**/



import java.io.*;
import gov.nih.nlm.nls.skr.*;


public class SRInteractive
{
   public static void main(String args[])
   {
        GenericObject myIntSRObj = new GenericObject(200, "username",
"password");


        // REQUIRED FIELDS:
        //    -- Email_Address
        //    -- APIText
        //
        // NOTE: The maximum length is 10,000 characters for APIText.
The
        //       submission script will reject your request if it is
larger.
```

```
        myIntSRObj.setField("Email_Address", "user@mail.com");



            String param = "";
            for(int i=1; i< args.length; i++){
                  param += args[i] + " ";
            }
            param += "\r\n";


        System.out.println(param);
            StringBuffer buffer = new StringBuffer(param);


        String bufferStr = buffer.toString();
        myIntSRObj.setField("APIText", bufferStr);


        // Optional field, program will run default SemRep if not
specified


        myIntSRObj.setField("COMMAND_ARGS", "-D -E");


        // Submit the job request


        try
        {
           String results = myIntSRObj.handleSubmission();
               System.out.print("results:\n");
           System.out.print(results);
               PrintWriter writer = new PrintWriter("results.txt",
"UTF-8");
               writer.println(results);
               writer.close();
```

```
        } catch (RuntimeException ex) {
           System.err.println("");
           System.err.print("An ERROR has occurred while processing
your");
           System.err.println(" request, please review any");
           System.err.print("lines beginning with \"Error:\" above and
the");
           System.err.println(" trace below for indications of");
           System.err.println("what may have gone wrong.");
           System.err.println("");
           System.err.println("Trace:");
           ex.printStackTrace();
        } // catch
            catch (Exception ex){
                  ex.printStackTrace();
            }
   } // main
} // class SRInteractive
```